TAMPEREEN
AMMATTIKORKEAKOULU

SERIALISOINTI JA TALLENTAMINEN PROTOBUF-NET-FORMAATILLA C#-POHJAISESSA PELISSÄ

Oula Piisalo

Opinnäytetyö
Syyskuu 2016
Tietojenkäsittelyn koulutusohjelma
Pelituotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Pelituotanto

PIISALO, OULA:
Serialisointi ja tallentaminen Protobuf-net-formaatilla C#-pohjaisessa pelissä

Opinnäytetyö 45 sivua, joista liitteitä 2 sivua
Syyskuu 2016

Opinnäytetyön tarkoituksena oli toteuttaa datan serialisointi- ja tallennusratkaisu toimeksiantajan peliin. Työn toimeksiantajana toimi tamperelainen pelialan yritys Dreamloop Games Oy. Alun perin yritys tarvitsi peliinsä datan serialisointi- ja tallennusratkaisun, joka toimisi mahdollisimman hyvin verkkopeliympäristössä. Serialisointiformaatiksi valittiin Protobuf-net sen nopeuden ja tehokkuuden vuoksi.

Opinnäytetyön teoriaosuudessa käsitellään keskeisiä tekniikoita ja työkaluja, jotka vaikuttivat serialisointiratkaisun kehittämiseen. Unity3D on pelinkehitysalusta, jolla Challengers of Khalea on kehitetty. Kehityksessä on myös käytetty StrangeIoC-viitekehystä, joka sisältää MVCS-kontekstiarkkitehtuurin. Datan serialisointi ja tallentaminen toteutettiin erillisenä moduulina – niin kutsuttuna palveluna.

Protobuf-net-palvelusta muodostui helposti käytettävä ja selkeä moduuli. StrangeIoC-viitekehyksen ansiosta palvelun rakentaminen ja testaaminen onnistuivat erillään muusta ohjelmasta. Palvelun käyttäjän tarvitsee vain tietää serialisoitavan tai deserialisoitavan luokan tyyppi. Projektin edetessä uusia tyyppejä tulee varmasti, jolloin palvelua on laajennettava. Laajentaminen on tehty mahdollisimman yksinkertaiseksi dokumentoinnin ja esimerkkien avulla.

Opinnäytetyö vertailee myös eri serialisointiformaattien tehokkuutta ja nopeutta. Protobuf-net-formaatin lisäksi vertailuissa ovat Unity3D-pelinkehitysalustan binääri-formaatti, Newtonsoft JSON ja XML. Serialisointiformaattien suorituskykytestit kirjoitettiin C#-ohjelmointikielellä ja testien ajamiseen käytettiin Unity3D:tä. Nopeutta mitattiin formaattien serialisoinnissa ja deserialisoinnissa. Tehokkuus mitattiin sillä, kuinka suuri tiedostokoko oli serialisoidulla datalla levylle tallennettaessa.

Suorituskykytestien tulokset puolsivat Protobuf-net-formaattia. Se oli nopein sekä serialisoinnissa että deserialisoinnissa, ja lopullinen tiedostokoko tallennetulle tiedostolle oli pienempi kuin muilla serialisointiformaateilla.

Asiasanat: mvcs-arkkitehtuuri, serialisointi, strangeioc, unity

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Game Development

PIISALO, OULA:
Serialization and Saving with the Protobuf-net Serialization Format in a C# Based Game

Bachelor's thesis 45 pages, appendices 2 pages
September 2016

The purpose of this thesis was to implement a data serialization and saving solution for a game called Challengers of Khalea. The client for the thesis is a Tampere based games industry company Dreamloop Games Inc. The company needed a solution for data serialization and saving that would work well in an online multiplayer environment. A serialization format called Protobuf-net was chosen due to its great speed and performance.

The theoretical section explores the tools and techniques used in the development of Challengers of Khalea. The game is being developed using the Unity3D game engine. The StrangeIoC framework and its MVCS extension are also part of the development.

The Protobuf-net service turned out to be a user-friendly and straightforward module. To use the service, one only needs to know the type of the class which is serialized or deserialized. The service is also easily extensible due to thorough documentation and clear code examples that the user may follow.

In addition to the serialization solution presented here, this thesis contains speed and performance benchmarks for four different data serialization formats. The benchmarked formats were Protobuf-net, Unity3D binary format, Newtonsoft JSON and XML. Speed was measured in the time it took to serialize and deserialize data samples. In this context performance means how efficiently the serialization format compresses the data. The readability aspect of the serialization formats was also addressed.

In the benchmarks done for this thesis Protobuf-net had the highest performance. Protobuf-net had the best speeds for both serialization and deserialization and also compressed the given data most efficiently.

Key words: mvcs architecture, serialization, strangeioc, unity

SISÄLLYS

ERITYISSANASTO

attribuutti	Olio-ohjelmoinnissa luokan ilmentymän tietokenttä, johon tieto tallennetaan.
arkkitehtuuri (ohjelmointi)	Jokin vallitseva ajatus, jonka mukaan asioita toteutetaan. Esimerkiksi MVC-arkkitehtuuri.
moduuli	Itsenäinen, muista riippumaton ohjelman osa. Voidaan lisätä tai poistaa ilman, että se vaikuttaa muun ohjelman toimivuuteen.
modulaarinen rakenne	Moduuleista koostuva rakenne. Katso moduuli.
Unity	Unity tai Unity3D on pelinkehitysalusta, jonka on kehittänyt Unity Technologies.
olio	Olio-ohjelmointiin liittyvä käsite. Olio on luokan ilmentymä.
skripti	Unity-pelinkehitysalustalla käytettävä komponentti, joka on kirjoitettu C#- tai UnityScript-ohjelmointikielellä. Tässä opinnäytetyössä kaikki skriptit on kirjoitettu C#-kielellä.

1 JOHDANTO

Datan serialisoinnille on ollut tarvetta yhtä kauan kuin tietoverkkoja on ollut olemassa. Jotta data on voitu lähettää verkon yli vastaanottajalle, on se täytynyt saada siirrettävään muotoon. Tähän ongelmaan pureuduttiin alussa käyttämällä binääriprotokollia, joissa serialisoitu data ei ole luettavissa ihmiselle. Protokollat olivat myös aina vain tiettyyn tarpeeseen tehtyjä. Datan lähettämistä ja vastaanottamista varten datan lähettäjän ja vastaanottajan täytyi tehdä sopimus siitä, missä tietokentät sijaitsivat ja mitä ne sisälsivät. Näistä käytännöistä kuitenkin poistuttiin, kun standardeja alkoi ilmaantua 1980-luvulla. XML-serialisointiformaatti syntyi vuonna 1996, kun Internet alkoi kasvaa räjähdysmäisesti. (Binstock 2013.)

Vanhat standardit eivät olleet ihmissilmälle luettavia ja yleisesti koettiin, että SGML-metakielen kanssa sopusoinnussa olevalle formaatille olisi käyttöä. SGML on vuonna 1986 standardoitu metakieli, josta HTML-ohjelmointikielikin on peräisin. 2000-luvun alkuun mennessä XML olikin jo laajalti käytetty formaatti ja kaikki isoimmat ohjelmointikielet sisälsivät XML-kirjastot. (Binstock 2013.)

JavaScript-ohjelmointikielen yleistymisen myötä JSON-serialisointiformaatin käyttö on kasvanut suuresti. JSON-formaatti (JavaScript Object Notation) noudattaa JavaScriptin merkintäkieltä ja sitä voidaankin lukea suoraan JavaScript-kielenä. Sen etuna on myös sen laaja tuki useine työkaluineen ja kirjastoineen. Suurimpaan osaan ohjelmointikielistä saa JSON-kirjastoja, mikä tekee sitä helpon tavan siirtää tietoa hyvinkin erilaisten järjestelmien välillä. (Binstock 2013.)

JSON ei kuitenkaan sovellu kaikkiin tarkoituksiin hyvin, kuten tietokantoihin, sillä siitä uupuu muutamia haluttuja ominaisuuksia. Esimerkiksi kaksi puuttuvaa ominaisuutta ovat päivämäärä-datatyyppi ja kommentoinnin mahdollisuus. Näiden puutteiden johdosta on kuitenkin syntynyt uusia vaihtoehtoja kuten BSON, joka on binääri JSON-formaatti. Sen on keksinyt MongoDB Inc. ja sitä käytetäänkin yrityksen päätuotteessa, joka on avoimen lähdekoodin NoSQL-tietokanta MongoDB. (Binstock 2013.) Päiväyksellä 24.8.2016 MongoDB on listattu tietokantajärjestelmiä listaavan sivuston DB-Engines.com:in mukaan neljänneksi käytetyimmäksi tietokantajärjestelmäksi (DB-Engines 2016).

Turhautuminen JSON-formaattiin on myös kannustanut kokeilemaan ja etsimään täysin uusia ratkaisuja. Git-versionhallintapalveluita tarjoavan sivusto GitHub:in yksi perustajajäsenistä, Tom Preston-Werner, on kehittänyt formaatin nimeltä TOML, joka on lyhenne sanosita Tom's Obvious, Minimal Language. Kehittäjänsä mukaan nimetty formaatti on yhtä kompakti kuin JSON, vaikka se käyttää erilaista merkintäskeemaa. TOML ei kuitenkaan ole ainoa vaihtoehto. Esimerkiksi teknologiayritys Google on kehittänyt kevyen ja nopean formaatin alun perin Java-, Python- ja C++-kielille, ja tuki monille muillekin kielille on kehitteillä. Formaattia kutsutaan protokollapuskureiksi (Protocol Buffers) ja Google käyttää kyseistä formaattia paljon omissa projekteissaan. (Binstock 2013; Google 2016.)

Kysymys-vastaus-sivusto StackOverflow.com:in ahkera käyttäjä, isobritannialainen ohjelmoija Marc Gravell on kehittänyt Googlen protokollapuskureiden pohjalta oman formaatin ohjelmistoyritys Microsoftin .NET kehitysympäristöön. Formaatti on yhteensopiva lähes kaikkien .NET-perheeseen kuuluvien ohjelmointikielien kanssa. Gravell on nimennyt formaatin Protobuf-net:ksi. (Gravell 2016)

Opinnäytetyön tarkoituksena on toteuttaa Protobuf-net-serialisointiformaattia hyödyntävä palvelu toimeksiantajan peliin. Tavoitteena on tehdä palvelusta mahdollisimman selkeä, toimiva ja yksinkertainen. Opinnäytetyön tavoitteena on myös selvittää, kuinka suorituskykyinen serialisointiformaatti Protobuf-net on verrattuna muihin yleisesti käytettyihin formaatteihin. Suorituskykytestien tulokset eivät kuitenkaan vaikuta käytettävään serialisointiformaattiin toimeksiantajan pelissä Challengers of Khalea. Testien tarkoituksena on havainnollistaa eroja serialisointiformaattien välillä.

2 DREAMLOOP GAMES OY

2.1 Yrityksen taustat

Dreamloop Games Oy on tamperelainen pelialan startup-yritys. Yrityksen perustivat Joni Lappalainen (toimitusjohtaja), Steve Stewart (markkinointijohtaja) ja Hannes Väisänen (teknologiajohtaja) vuonna 2015. Yrityksen esikoispeli, Challengers of Khalea, on ollut kehitteillä jo kuitenkin syksystä 2014 alkaen. Vuoden 2016 alusta Dreamloop Games Oy yhdistyi toisen tamperelaisen pelialan yrityksen, Vasara Entertainment, kanssa. Vasara Entertainment on julkaissut yhden pelin vuonna 2015. (Dreamloop Games. 2016.)

2.2 Challengers of Khalea

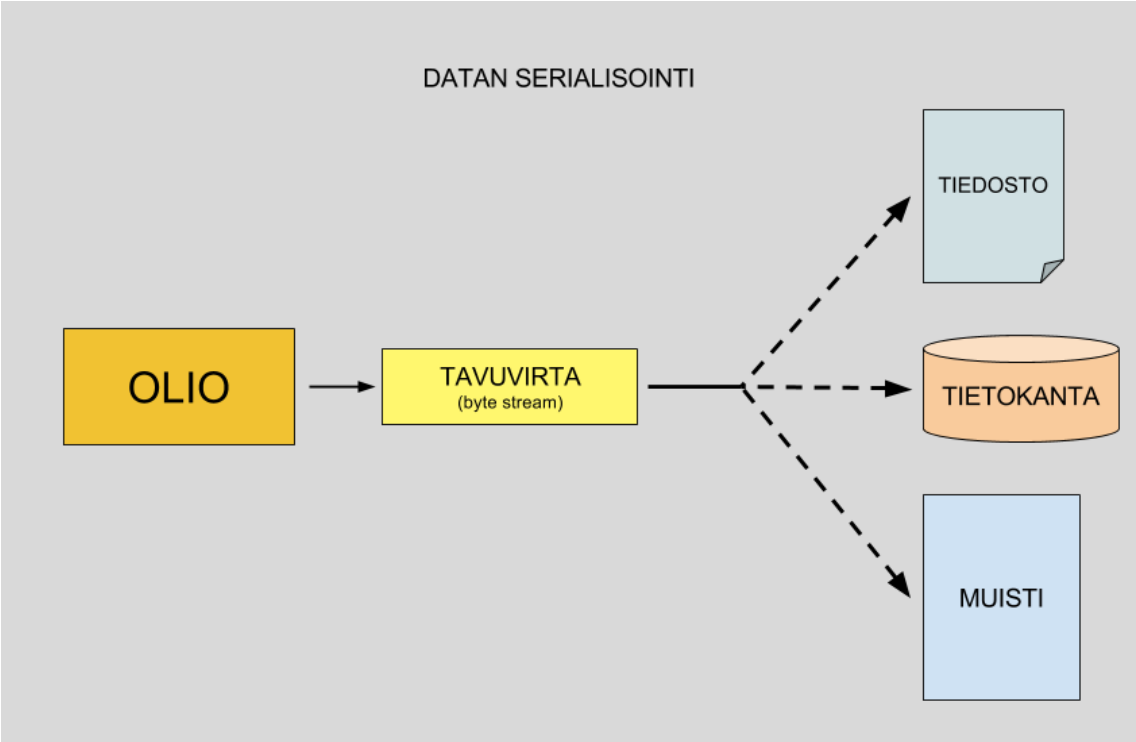
Challengers of Khalea yhdistää suosittujen jalkapallomanagerointipelien hallinnoinnin ja vuoropohjaisten strategiapelien taktiset elementit yhdeksi kokonaisuudeksi. Peli sijoittuu Khalean taianomaiseen maailmaan, jossa suosituin urheilulaji on joukkueiden väliset gladiaattoriottelut. Pelaaja toimii pelissä gladiaattorijoukkueen managerina ja tarkoituksena on johdattaa joukkue Khalean mestareiksi.

Challengers of Khalea oli alun perin suunniteltu verkossa pelattavaksi moninpeliksi, mutta suuntaa päätettiin muuttaa kesken kehityksen. Pelistä päätettiin tehdä yksinpeli, joka ei vaadi Internet-yhteyttä toimiakseen. Peli koostuu kahdesta pelillisestä osa-alueesta: vuoropohjaisesta strategiasta ja joukkueen manageroinnista. Gladiaattoriottelut ovat vuoropohjaista strategiaa, joissa pelaaja käskyttää viittä valitsemaansa joukkueensa gladiaattoria. Gladiaattorien tarpeita on täytettävä myös otteluiden ulkopuolella: gladiaattorit ovat palkattava ja heille on hankittava tarvittavat varusteet. Joukkueeseen kuuluu ottelevien gladiaattorien lisäksi valmentaja, joka tuo oman osuutensa otteluihin, ja huoltohenkilöitä. Huoltohenkilöihin kuuluvat muun muassa hieroja, joka auttaa gladiaattoreita palautumaan otteluiden jälkeen. Voittaakseen pelin, on pelaajan noustava ylimpään gladiaattoriliigaan ja voitettava se. Challengers of Khalea on saanut paljon vaikutteita suomalaisesta Areena-tietokonepelisarjasta.

3 DATAN SERIALISOINTI

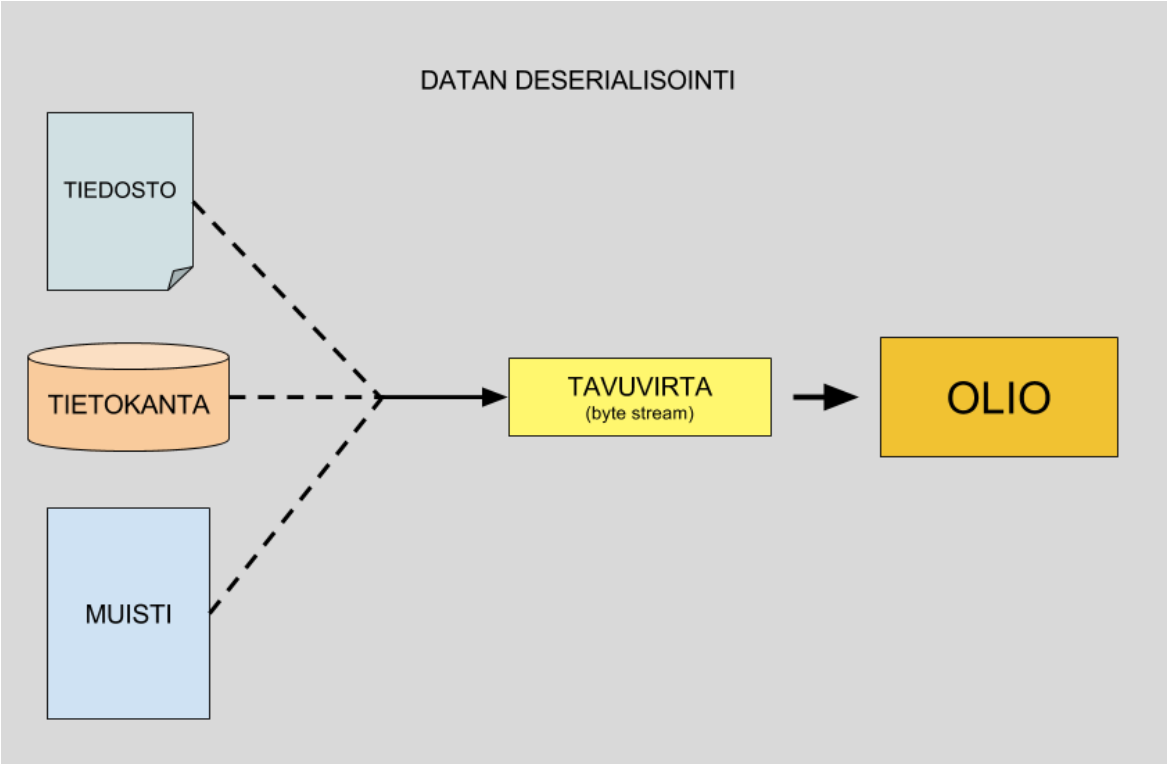
3.1 Serialisointi

Tietotekniikassa serialisointi (serialization) on menetelmä, jolla datarakenteen tai olion tila muunnetaan merkkijonoksi. Serialisoinnilla sovelluksen data saadaan peräkkäiseksi luetteloksi dataelementtejä. Datan on oltava binääri- tai tekstimuodossa, jotta se säilyisi tallennuslaitteella tai se voitaisiin siirtää verkon yli. (Saikkonen 2012.) Kuviossa 1 on havainnollistettu datan serialisointia olio-ohjelmoinnissa. Olio muutetaan (serialisoidaan) merkkijonoksi, jolloin se voidaan kirjoittaa tavuvirtaan. Kun olio on saatu tavuvirtaan, se voidaan tallentaa tietokantaan, levylle tiedostoksi tai muistiin.



Kuvio 1. Datan serialisointi

Deserialisointi on serialisoinnin vastakkainen menetelmä, jolla käännetään serialisoitu data takaisin ohjelmalle käytettävään muotoon. Kuviossa 2 on havainnollistettu datan deserialisointia. Data voidaan lukea tiedostosta, tietokannasta tai muistista tavuvirtaan, josta se serialisoidaan takaisin olioksi, joka se oli ennen serialisointia.



Kuvio 2. Datan deserialisointi

Serialisoinnista keskusteltaessa esiintyy usein termi marshalling, joka tarkoittaa suomennettuna jäsentämistä. Ohjelmointikielestä riippuen sitä käytetään serialisoinnin synonyymina. Esimerkiksi Python-ohjelmointikielessä serialisointi tarkoittaa samaa kuin jäsentäminen ja sille onkin keksitty Pythonille ominainen käsite pickling, joka viittaa suolakurkkujen säilömiseen. Vastaoperaatio on nimeltään unplickilng. Jotkin ohjelmointikielet kuitenkin erottavat nämä kaksi toisistaan. Java-ohjelmointikielen standardien mukaan jäsentäminen tarkoittaa olion tilan ja lähdekoodin rungon tallentamista. Javassa serialisointiin ei kuulu lähdekoodin tallentamista. (Ryan, Seligman & Lee 1999.)

3.2 Pelin tarpeet datan serialisoinnille

Challengers of Khalea sisältää paljon erilaisia ennalta suunniteltuja malleja, joiden kanssa pelaaja on tekemisissä. Ennalta suunniteltuja malleja luodaan niitä varten kehitetyillä muokkaustyökaluilla, jonka jälkeen ne serialisoidaan ja tallennetaan paikallisesti levylle. Pelin alkaessa kaikki suunnitellut mallit luetaan levyltä, deserialisoidaan ja asetetaan oi-

keille paikoilleen pelissä. Tämä mahdollistaa sisällön lisäämisen peliin ilman, että lähdekoodia tarvitsee muuttaa. Suunniteltuja malleja ovat osa gladiaattoreista, gladiaattorien varusteet, dialogit, ottelun aikana laukeavat tapahtumat, tarinaan liittyvät vastustajien joukkueet sekä paljon muita. Suunniteltujen mallien lisäksi pelissä on paljon pelin aikana luotavia malleja, jotka usein saavat satunnaiset arvot tietokenttiin. Esimerkiksi palkattavissa olevat gladiaattorit generoituvat automaattisesti eikä niitä suunnitella erikseen.

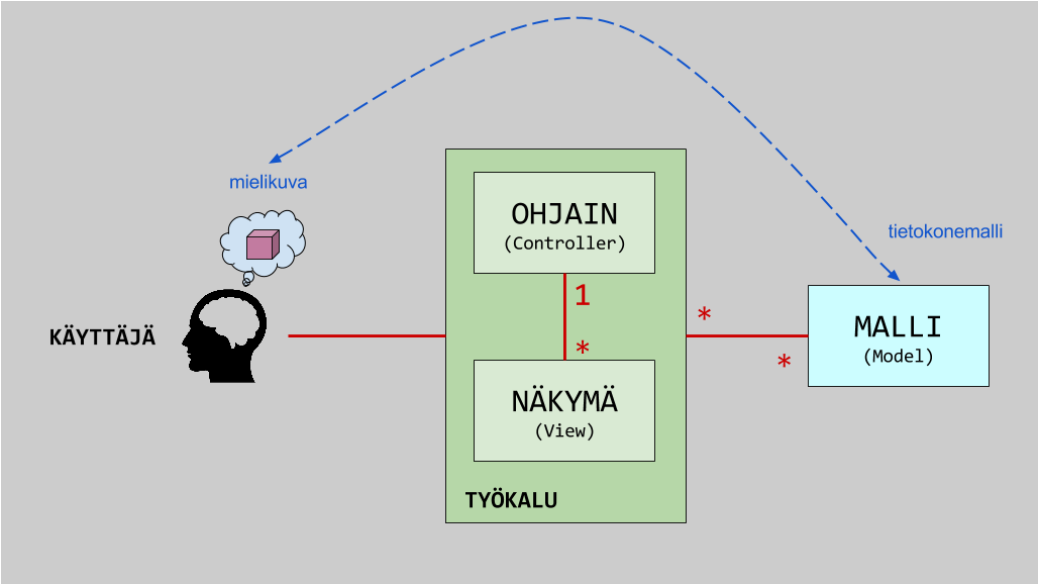
Jotta pelaaja voi jatkaa peliä myöhemmin, on pelin tilanne saatava tallennettua johonkin. Pelin tilanteen tallentaminen toteutetaan SQLite-tietokannan avulla eikä sen toteuttamiseen tarvita datan serialisointia, jota tässä opinnäytetyössä käsitellään.

4 KEHITYSTYÖSSÄ KÄYTETYT MENETELMÄT

4.1 MVC-arkkitehtuuri

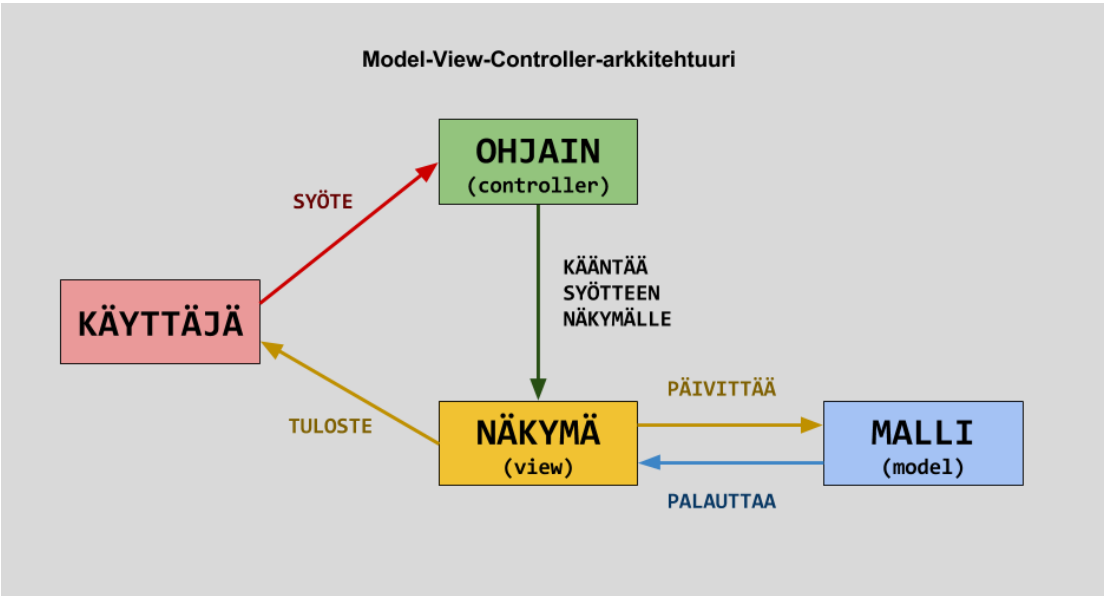
MVC on ohjelmistoarkkitehtuuri, jonka nimi muodostuu englanninkielen sanoista model, view ja controller. MVC-arkkitehtuurin on alun perin muotoillut norjalainen professori Trygve Reenskaug vuosina 1978-79 vieraillessaan Palo Alton tutkimuskeskuksessa. (Reenskaug 1979.) MVC:n ja sen variaatioiden perusideana on pilkkoa ohjelma useampaan eri osa-alueeseen esimerkiksi erottamalla datan käsittely -logiikan sen näyttämislogiikasta. Tavallisen MVC-arkkitehtuurin mukaan osa-alueita on kolme: *malli* (model), *näkymä* (view) ja *ohjain* (controller).

MVC-arkkitehtuurista löytyy monia erilaisia lähestymistapoja. Tässä esitetty MVC-arkkitehtuurin malli noudattaa Reenskaugin (1979) dokumentaation esittelemää muotoa. Kuvio 4 havainnollistaa MVC-arkkitehtuurin keskeistä tarkoitusta kaventaa kuilua ihmiskäyttäjän mielikuvan ja tietokoneen digitaalisen mallin välillä. Malli kuvastaa tietoa ja se voi olla yksi olio tai olioista koostuva rakenne. Mallissa oleva tieto esitetään käyttäjälle näkymien kautta, jotka toimivat niin kutsuttuna esityssuodattimena. Näkymä on kiinnitettynä sen malliin (tai osaan mallista) ja se saa datan näyttämiseen tarvittavat tiedot kysymällä ne mallilta. Se voi myös päivittää mallia lähettämällä täsmällisiä viestejä. Kaikkien kyselyiden ja viestien täytyy käyttää samaa terminologiaa mallin kanssa. Näkymän on siis tiedettävä mallin attribuuttien merkitykset. (Reenskaug 1979.) Samalla mallilla voi olla – ja useimmiten onkin – useampia näkymiä ja yksi näkymä voi esittää useamman mallin dataa.



Kuvio 4. MVC-arkkitehtuurin keskeinen tarkoitus on kaventaa kuilua käyttäjän mielikuvan ja tietokoneen digitaalisen mallin välillä. (Reenskaug 1979. muokattu)

Reenskaugin periaatteen mukaan ohjaimen tehtävä on toimia käyttäjän ja ohjelman välisenä linkkinä tarjoten käyttäjälle tarvittavat näkymät ja kontrollit. Ohjain käsittelee myös käyttäjän antamat syötteet ja muuntaa ne sopiviksi viesteiksi halutuille näkymille. Ohjaimen ei kuitenkaan tulisi koskaan täydentää näkymiä esimerkiksi yhdistämällä useampaa näkymää toisiinsa. Reenskaugin mallin mukaan näkymän ei myöskään tulisi koskaan tietää käyttäjän syötteistä kuten hiiren toiminnoista tai näppäinten painalluksista. Käyttäjä ei suoraan voi vaikuttaa mallissa olevaan dataan vaan ohjaimet kuuntelevat käyttäjän syötteitä ja kääntävät ne eteenpäin näkymälle, joka päivittää ne malliin. Kuviossa 3 on esitettynä MVC-arkkitehtuurin toimintamalli, jonka Reenskaug kehitti vuosina 1978-1979.



Kuvio 3 Trygve Reenskaugin mukainen malli MVC-arkkitehtuurista.

Jos Reenskaugin mukaista MVC-arkkitehtuurimallia sovellettaisiin Challengers of Khaleaan, tarkoittaisi malli esimerkiksi pelissä olevan gladiaattorin ominaisuuksia ja niiden arvoja. Esimerkiksi voimaa, nopeutta ja palkkaa. Jos pelihahmo ottaa vastaan iskun, ohjain vie tästä tiedon näkymään, johon se tekee tarvittavat muutokset kuten elämäpisteiden väheneminen. Näkymä päivittää elämäpisteiden muutokseen gladiaattorin digitaaliseen malliin

MVC-arkkitehtuuria sovelletaan käyttötarkoituksista riippuen eri tavoilla. Monet kehittäjät ovat poikenneet Reenskaugin luomasta mallista ja määrittäneet erilaiset suhteet MVC-komponenteille. Esimerkiksi joissain lähestymistavoissa ohjain on vastuussa mallin ja näkymän välisestä tiedonsiirrosta, jolloin ne eivät ole kytköksissä toisiinsa. (Google. 2016a)

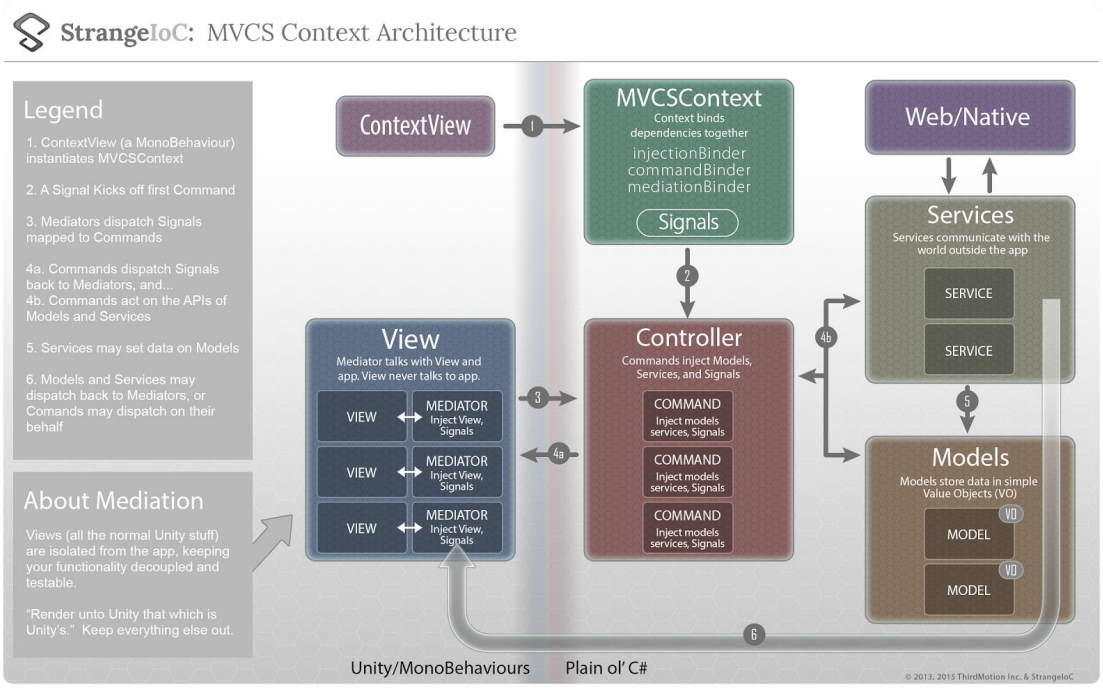
4.2 StrangeIoC

Toimeksiantajan projektissa Challengers of Khalea ei alkuun käytetty viitekehyksiä tai arkkitehtuureja, joiden mukaan sitä olisi kehitetty. Projektin kasvaessa ja edetessä huomattiin kuitenkin, että jonkinlainen järjestys ja rakenne olisi hyvä olla. Viitekehykseksi valittiin StrangeIoC, jonka tarkoituksena on tehdä koodista modulaarisempaa ja helposti testattavaa.

StrangeIoC on hallinnan muutossuunnan vaihtamiseen (engl. Inversion of Control, IoC) ja datan sitomiseen (binding) tarkoitettu viitekehys, joka on kirjoitettu nimenomaan C#-ohjelmointikielelle ja Unity-pelikehitysalustalle. Sen kehittäjä on ThirdMotion, Inc. ja se perustuu pitkälti ActionScript 3.0:lle kehitettyyn avoimen lähdekoodin Robotlegs-viitekehykseen. StrangeIoC ei kuitenkaan ole sovitus Robotlegsistä, mutta sen piirteitä on käytetty runsaasti StrangeIoC:ssä. (ThirdMotion, Inc. & StrangeIoC 2015b.) StrangeIoC sisältää myös valinnaisen MVCS-kontekstiarkkitehtuurirakenteen, jota hyödynnetään toimeksiantajan projektissa.

StrangeIoC-viitekehyksen MVCS-kontekstiarkkitehtuuri on suunniteltu toimivan MVC-arkkitehtuurin tapaan. MVCS-kontekstiarkkitehtuurissa kirjain ”S” tarkoittaa palvelua

(engl. service). Palvelu on usein jokin mikä tahansa toiminto, joka viittaa ohjelman ulkopuolelle kuten vaikkapa web-palvelu. (Third Motion, Inc & StranceIoC 2015a.) Challengers of Khaleassa käytetään useita palveluja, joista yksi on tässä opinnäytetyössä käsiteltävä Protobuf-net-serialisointipalvelu. Vaikka StrangeIoC:n MVCS-kontekstiarkkitehtuurin mukaan kehitetyt ohjelmat on suunniteltu toimimaan MVC:n periaatteiden mukaisesti, eroaa se jonkin verran Reenskaugin MVC-arkkitehtuurista. Kuvio 5 on Third Motion Inc:n luoma visuaalinen kaava StrangeIoC:n käyttämästä MVCS-kontekstiarkkitehtuurista ja sen eri komponenttien välisestä toiminnasta. Esitetyssä mallisa näkymä on pilkottu kahteen osaan, joista toista kutsutaan välittäjäksi (mediator). Välittäjän tehtävä on välittää tietoa muusta ohjelmasta näkymälle. Hyvien käytäntöjen mukaan näkymän ei käytännössä tarvitse tietää mitään muusta ohjelmasta. Komponentit lähettävät niin kutsuttuja signaaleja välittääkseen tietoa. Kun signaali lähetetään, ei lähettäjä tiedä sen vastaanottajaa. Signaalin vastaanottaja, kuuntelija, käyttää signaalissa olevia tietoja hyväkseen ja tekee sen perusteella tarvittavat toimenpiteet. Välittäjä voi esimerkiksi kuunnella ChangeColorSignal-signaalia, jolloin sen lauetessa välittäjä vaihtaa näkymänsä väriarvoja.



Kuvio 5. StrangeIoC MVCS-konteksti-arkkitehtuuri. Kuva löytyy isompana liitteistä. (ThirdMotion, Inc & StrangeIoC 2015c.)

StrangeIoC tarjoaa datansidontaviitekehyksen (binding framework), jonka avulla on mahdollista sitoa (yhdistää) yksi tai useampi asia yhteen tai useampaan johonkin toiseen asiaan. Esimerkiksi kaksi luokkaa voidaan sitoa toisiinsa ja kun toinen niistä luodaan,

syntyy toinenkin automaattisesti. Tämä on hyvin käytännöllistä varsinkin rajapinta-luokkien tapauksessa. Rajapintaluokka voidaan sitoa luokkaan, joka toteuttaa sitä. Kun tämän jälkeen halutaan käyttää edellä mainittua luokkaa, kutsutaankin suoraa sen rajapintaluokkaa. Sidoksen ansiosta rajapintaa toteuttava luokka syntyy automaattisesti. Sidontaa käytetään StrangeIoC-viitekehyksessä rajapintaluokkien lisäksi näkymien ja välittäjien sitomisessa sekä signaalien ja komentojen sitomisessa. (ThirdMotion, Inc. & StrangeIoC 2015b.)

Hallinnan muutossuunnan vaihtaminen (engl. Inversion of Control, IoC) on suunnitteluperiaate, jonka avulla saadaan vähennettyä ohjelmiston luokkien välisiä riippuvuuksia. Periaate on hyvin yksinkertainen: koodimoduulien välisten riippuvuuksien abstrahointi ja niiden riippuvuuksien tyydyttäminen ohjelman ajon aikana. (StrangeIoC 2015.) Tähän periaatteeseen viitataan usein leikkimielisesti ”Hollywood periaatteella” toisin sanoen ”älä soita meille, me soitamme sinulle”. (Fowler 2005.) Hallinnan muutossuunnan vaihtamista varten StrangeIoC tarjoaa riippuvuusinjektoinnin (dependency injection). Riippuvuusinjektiolla oliolle annetaan (injektoidaan) sen ilmentymän (instanssin) muuttujat sen sijaan, että olio määrittäisi ne itse. (Shore 2006.) Tarkoituksena on luoda niin sanotusti ”väljästi paritettuja” luokkia, joissa ei suoraan viitata toisiin luokkiin. Väljästi paritetun vastakohta on ”tiukasti paritettu”, jolloin luokat viittaavat suoraan toisiinsa. Koodiesimerkissä 1 on esitetty tiukasti paritettujen luokkien suhde. Luokka TighlyCoupled sisältää MyModel-luokan ilmentymän, jolloin ne ovat paritettu tiukasti toisiinsa.

```
public class MyModel
{
	public int x;
	public int y;
}

public class TighlyCoupled
{
	MyModel myModel = new MyModel ();

	void Start()
	{
		myModel.x = 4;
		myModel.y = 5;
	}
}
```

Koodiesimerkki 1. Tiukasti paritetut luokat

Koodiesimerkissä 2 on esitetty ”väljästi paritettu” -luokka StrangeIoC:n ja MVCS-kontekstin avulla. Sen sijaan, että LooselyCoupledExampleCommand-luokka sisältäisi

suoraan MyModel-luokan ilmentymän, se viittaa MyModel-luokan rajapintaan IMyModel käyttämällä StrangeIoC:n riippuvuusinjektiota. IMyModel-rajapintaluokka sidotaan MyModel-luokkaan MyContext-kontekstissa. Sitomisen ansiosta MyModel-luokka luodaan aina kun rajapintaa IMyModel käytetään.

```
public interface IMyModel
{
	int x { get; set; }
	int y { get; set; }
}

public class MyModel : IMyModel
{
	public int x { get; set; }
	public int y { get; set; }
}

public class MyContext : MVCSContext
{
	…
	protected override void mapBindings()
	{
		injectionBinder.Bind<IMyModel>().To<MyModel>();
	}
	…
}

public class LooselyCoupledExampleCommand
{
	[Inject]
	public IMyModel MyModel { get; set; }

	protected override Execute()
	{
		MyModel.x = 4;
		MyModel.y = 5;
	}
}
```

Koodiesimerkki 2. Väljästi paritetut luokat StrangeIoC:n ja MVCS-kontekstin avulla

5 PROTOKOLLAPUSKURIT JA PROTOBUF-NET

5.1 Protokollapuskurit

Protokollapuskurit (protocol buffers) ovat teknologiayritys Googlen kehittämä tapa serialisoida strukturoitua dataa. Ne toimivat lähes riippumattomina ohjelmointikielestä ja kehitysalustasta. Protokollapuskurit ovat myös laajennettavissa. Googlen protokollapuskurit tukevat tällä hetkellä Java-, Python-, C++-, Go-, JavaNano-, Ruby- ja C#-ohjelmointikieliä. Jotta protokollapuskurit toimisivat, käyttäjän täytyy määritellä ensimmäiseksi serialisoitavien tietojen rakenne .proto-tiedostoihin. Jokainen protokollapuskuri-viesti on pieni looginen tallenne, joka sisältää nimi–arvo-parin. Jokaisella viestityypillä on yksi tai useampi uniikisti numeroitu kenttä, joilla on nimi ja arvon tyyppi. Arvotyypit voivat olla numeroita (kokonais- tai liukulukuja), totuusarvoja, merkkijonoja, raakatavuja tai jopa toisia protokollapuskuri -viestityyppejä, jotka mahdollistavat käyttäjän jäsentää datan hierarkkisesti. (Google 2016b.) Liite 2 on esimerkki protokollapuskurien käyttämästä .proto-tiedostosta.

5.2 Protobuf-net-serialisointiformaatti

Protobuf-net-serialisointiformaatti tallentaa datan Googlen kehittämien protokollapuskurien avulla. Ohjelmointirajapinta on kuitenkin erilainen verrattuna Googlen protokollapuskureihin. Protobuf-net on kehitetty .NET-viitekehykselle ja sen ohjelmointirajapinta noudattaa yleistä .NET mallia ja on käytännössä verrattavissa XmlSerializer- ja DataContractSerializer-serialisointiformaatteihin. Protobuf-netin kehittäjän, Marc Gravelin, mukaan sen pitäisi toimia useimmilla .NET-ohjelmointikielillä, jotka kirjoittavat perustyyppejä ja käyttävät attribuutteja. (Gravell 2016.) Gravell ei ole luonut kehittämälleen serialisointiformaatille kovinkaan kattavaa dokumentaatiota. Tätä paikkaa kuitenkin se, että hän on todella ahkera käyttäjä ohjelmointiaiheisiin kysymyksiin erikoistuneella kysymys-vastaus-sivustolla StackOverflow.com:ssa.

Googlen kehittämien protokollapuskurien tapaan Protobuf-net voi käyttää .proto-tiedostoja ja luoda niiden pohjalta serialisoitavat luokat. Protobuf-net tukee kuitenkin myös niin kutsuttua luokkien somistamista. Luokkiin ja niiden tietokenttiin merkitään tarvittavat

tunnisteet Protobuf-net-serialisointia varten. Luokkien somistamista on havainnollistettu alaluvussa 7.2.

6 SUORITUSKYKYTESTIT

6.1 Lähtötilanne

Useimmissa ohjelmointikielissä datan serialisointiin löytyy paljon erilaisia formaatteja. C# ei ole poikkeus. C#:ssa on valmiiksi sisäänrakennettuna binääri- ja XML-serialisointiformaatit ja näiden lisäksi löytyy useita kolmansien osapuolien kehittämiä kirjastoja.

Tässä osiossa suorituskykytestien tarkoituksena on vertailla eri serialisointitapoja kahdella eri osa-alueella: nopeus ja tehokkuus. Nopeus mitataan ottamalla aikaa, kuinka kauan datan serialisoimisen ja levylle tallentaminen kestää. Nopeus mitataan myös vastaoperaatiossa, joka tässä tapauksessa tarkoittaa datan levyltä lukemisesta ja datan deserialisointia. Tehokkuus testeissä tarkoittaa serialisoidun datan tiedostokokoa. Pienempi tiedostokoko tarkoittaa tehokkaampaa serialisointia. XML-, JSON- ja binääriserialisointiformaatit ovat valittu suorituskykytestiin Protobuf-net-formaatin rinnalle, koska ne löytyvät Unity-pelinkehitysalustan dokumentaatiosta tai ovat sisäänrakennettuna C#-kieleen. JSON-serialisointiin käytetään Newtonsoftin kehittämää kirjastoa .NET-viitekehystä käyttäville ohjelmointikielille.

Suorituskykytestit suoritettiin viidellesadalle (500) luodulle SerializableModel-luokalle. Serialisolinti- ja deserialisointitestit suoritetaan molemmat sata (100) kertaa. Testien tulokset esitetään tekstimuodossa sekä visuaalisina diagrammeina. Koodiesimerkki 3 on SerializableModel-luokka, jota käytetään testeissä. Luokan instansseille annetaan satunnaiset arvot kaikkiin kokonaislukukenttiin paitsi Id-kenttään, joka on luokan tunnistenumero. Se määritetään juoksevana lukuna.

```
using System;
using System.Runtime.Serialization;
using ProtoBuf;

[ProtoContract, Serializable, DataContract]
public class SerializableModel
{
    [DataMember, ProtoMember(1)] public string Name { get; set; }
    [DataMember, ProtoMember(2)] public int Id { get; set; }
    [DataMember, ProtoMember(3)] public int Gender { get; set; }
    [DataMember, ProtoMember(4)] public int Age { get; set; }
    [DataMember, ProtoMember(5)] public int Constitution { get; set; }
    [DataMember, ProtoMember(6)] public int Strength { get; set; }
    [DataMember, ProtoMember(7)] public int Agility { get; set; }
    [DataMember, ProtoMember(8)] public int Endurance { get; set; }
    [DataMember, ProtoMember(9)] public int Intelligence { get; set; }
    [DataMember, ProtoMember(10)] public int Reaction { get; set; }
    [DataMember, ProtoMember(11)] public int Stamina { get; set; }
    [DataMember, ProtoMember(12)] public int Mobility { get; set; }
    [DataMember, ProtoMember(13)] public int Marksmanship { get; set; }
    [DataMember, ProtoMember(14)] public int Experience { get; set; }
    [DataMember, ProtoMember(15)] public int AxeSkill { get; set; }
    [DataMember, ProtoMember(16)] public int BluntSkill { get; set; }
    [DataMember, ProtoMember(17)] public int BowSkill { get; set; }
    [DataMember, ProtoMember(18)] public int CrossbowSkill { get; set; }
    [DataMember, ProtoMember(19)] public int MagicSkill { get; set; }
    [DataMember, ProtoMember(20)] public int PolearmSkill { get; set; }
    [DataMember, ProtoMember(21)] public int ShieldSkill { get; set; }
    [DataMember, ProtoMember(22)] public int SwordSkill { get; set; }
    [DataMember, ProtoMember(23)] public int SalaryWish { get; set; }
}
```

Koodiesimerkki 3. Suorituskykytesteissä käytettävä serialisoitava luokka.

Testit ovat suuntaa antavia eikä välttämättä tehokkain tai nopein serialisointitapa ole paras kaikissa tilanteissa. Koska Challengers of Khalea -peliä kehitetään Unity3D-pelinkehitysalustalla, suorituskykytesti toteutettiin myös kyseisellä pelinkehitysalustalla yhtenäisyyden saavuttamiseksi. Testit suunniteltiin ja suoritettiin Unity3D:n versiolla 5.3.5p5. Suorituskykytestit suoritettiin Windows 10 Pro -käyttöjärjestelmällä. Keskusyksikkönä toimi Intel i5-2500, joka toimi 3,30 gigahertzin kellotaajuudella Järjestelmässä oli 8 gigatavua DDR3 muistia. Massamuistina, johon serialisoidut tiedostot tallennettiin, käytettiin 640 gigatavuista kovalevyä (HDD), joka toimi 7200 kierroksen minuuttivauhdilla.

6.2 Testattavat formaatit

Suorituskykytesteissä on testattavina ja vertailtavina serialisointiformaatteina XML-, Newtonsoft JSON-, Unity3D:n binääri- ja Protobuf-net-formaatti. XML-formaatilla (Extended Markup Language) data serialisoituu tekstimuotoon. Se on suunniteltu sekä ihmiselle että koneelle luettavaksi. XML käyttää yleensä skeemaa määrittämään XML-

dokumentin rakenteen. JSON (Javascript Object Notation) on JavaScript-ohjelmointikielen pohjalta tehty serialisointiformaatti. JSON-formaatilla luokan kaikki kentät kääntyvät tekstimuotoon, joka on luettavissa ihmissilmälle. Koska serialisoitu data on vain tekstiä, voi mikä tahansa ohjelmointikieli lukea sitä. Tästä syystä JSON on hyvin suosittu formaatti datan serialisointiin. Serialisoitu olio muuttuu periaatteessa koodiksi, jonka JavaScript-tulkki voi lukea suoraan. Käytännössä lukeminen tehdään erillisen kirjaston avulla, vaikka kyseessä olisikin JavaScript-kieli. (Saikkonen 2012.)

Unity3D:n binääriformaatilla luokan dataelementit serialisoituvat binäärimuotoon, joka tarkoittaa siät, ettei data ole tarkoitettu luettavaksi ihmissilmälle. Protobuf-net-formaatilla luokka serialisoituu myös binäärimuotoon. Protobuf-net on sopimuspohjainen (data contract) formaatti, mikä tarkoittaa sitä, että jokainen serialisoitava muuttuja ja luokka on erikseen merkittävä.

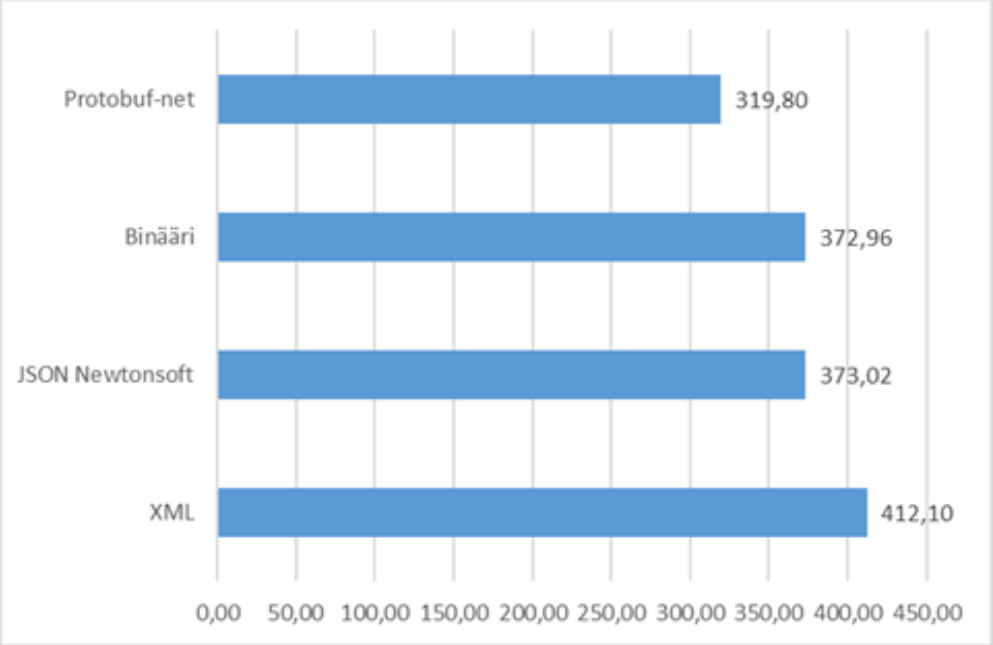
XML-, JSON- ja binääriserialisointiformaatit ovat valittu suorituskykytestiin Protobuf-net-formaatin rinnalle, koska ne löytyvät Unity3D-pelinkehitysalustan dokumentaatiosta tai ovat sisäänrakennettuna C#-kieleen. JSON-serialisointiin käytetään Newtonsoftin kehittämää kirjastoa .NET-viitekehystä käyttäville ohjelmointikielille.

6.3 Kirjoittaminen

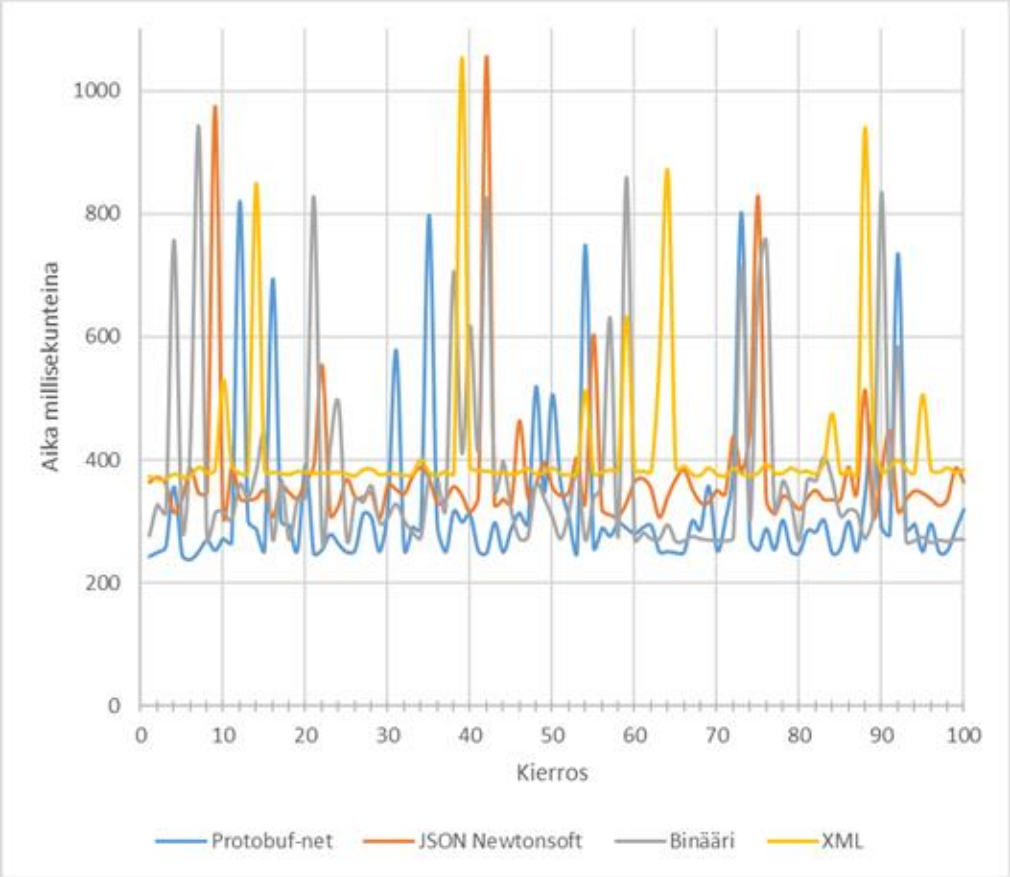
Tässä osiossa on purettu suorituskykytestien serialisointiosuus kirjallisesti sekä havainnollistavin kuvioin. Kuviot sisältävät vaaka- ja käyrädiagrammeja. Käyrissä on havaittavissa paljon piikkejä, jotka johtuvat todennäköisimmin käytetyn kovalevyn fyysisistä rajoitteista. Kirjoitettaessa pyörivän kovalevyn ulkoreunoille on nopeus suurempi ja kirjoitettaessa sisäreunoille on nopeus hitaampi.

Serialisointitestissä 500 luotua datamallia serialisoitiin ja kirjoitettiin levylle. Testi toistettiin 100 kertaa. Nopein formaatti testeissä oli Protobuf-net, joka serialisoi 500 mallia keskimäärin 319,80 millisekunnissa. Seuraavaksi nopein oli Unityn binääriserialisointi, jonka keskimääräinen serialisointiaika oli 372,96 millisekuntia. Kolmanneksi sijoittui JSON ajalla 373,02 millisekuntia. Hitain vertailtavista formaateista oli XML, jonka keskimääräinen serialisointiaika oli 412,10 millisekuntia. Binääri- ja JSON-formaatit olivat lähes tasoissa ja ne olivat Protobuf-net-formaattia 16,6 prosenttia hitaampia. XML-

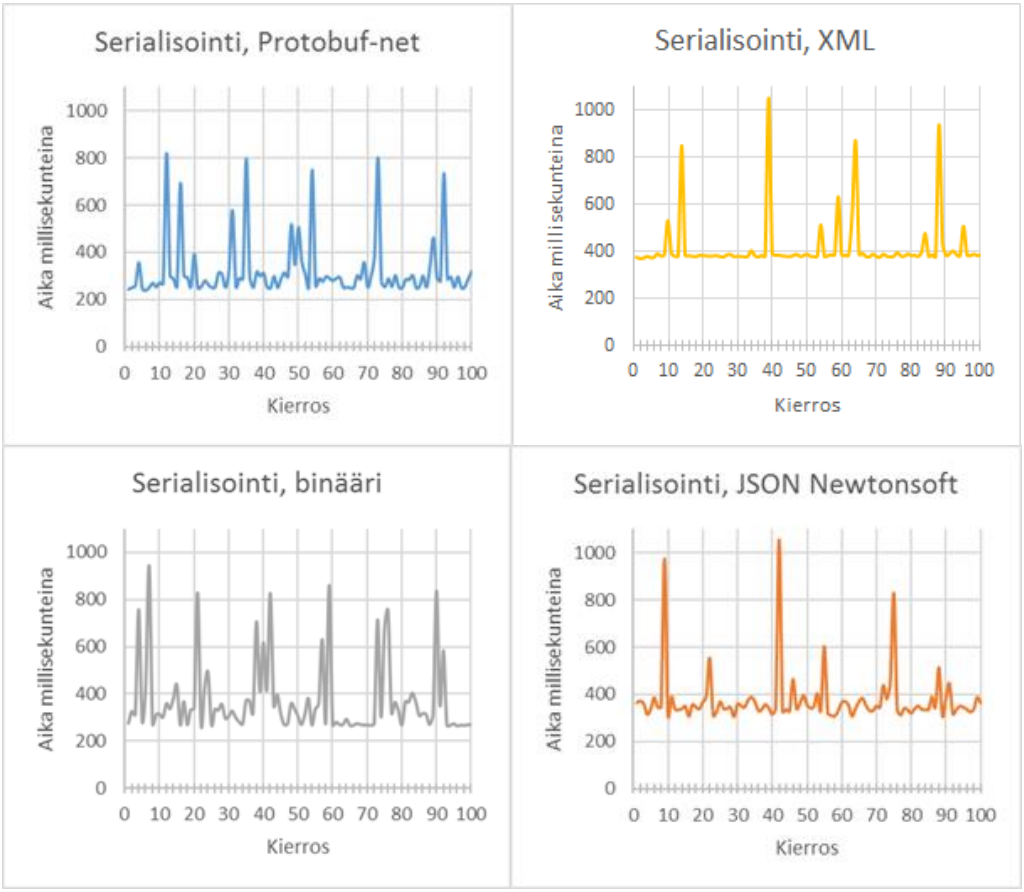
formaatti oli suhteutettuna Protobuf-net-formaattiin 28,9 prosenttia hitaampi. Kuviossa 6 on serialisointitestin tulokset purettuna visuaalisena diagrammina. Kuviossa 7 on havainnollistettuna serialisointitestien tulokset kaikki formaatit yhdistettynä ja kuviossa 8 tulokset ovat eriteltynä formaattien mukaan.



Kuvio 6. Keskimääräinen serialisointiaika millisekunteina.

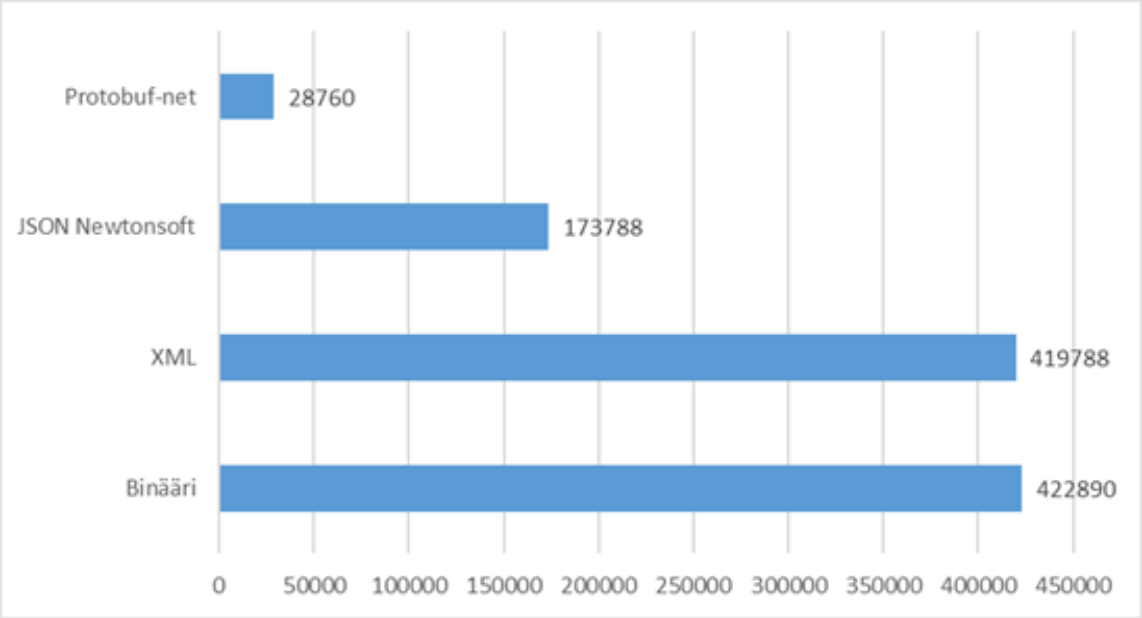


Kuvio 7. Serialisointitestien tulokset yhdistettynä.



Kuvio 8. Serialisointitestien tulokset eriteltynä formaatteihin.

Kun data oli serialisoitu, laskettiin levylle tallennettujen 500 tiedoston yhteiskoko jokaiselta formaatilta. Pienin yhteenlaskettu tiedostokoko saatiin aikaiseksi Protobuf-net-formaatilla, jolla se oli 28760 tavua. Toiselle sijalle testissä pääsi JSON Newtonsoft -, kolmanneksi XML- ja viimeiseksi Unity3D:n binääriformaatti. Protobuf-net-formaattiin suhteutettuna oli JSON-formaatilla tallennettu tiedosto 504,3 prosenttia suurempi. Toisin sanoen tiedostokoko oli yli kuusinkertainen. XML- ja binääriformaatti tallensivat tiedostot lähes saman kokoisina. Protobuf-net formaattiin suhteutettuna ne olivat 1359,6 ja 1370,4 prosenttia suurempia. Tämä tarkoittaa 14-kertaista tiedostokokoa. Kuviossa 9 on havainnollistettuna formaattien erot vaakadiagrammeina.

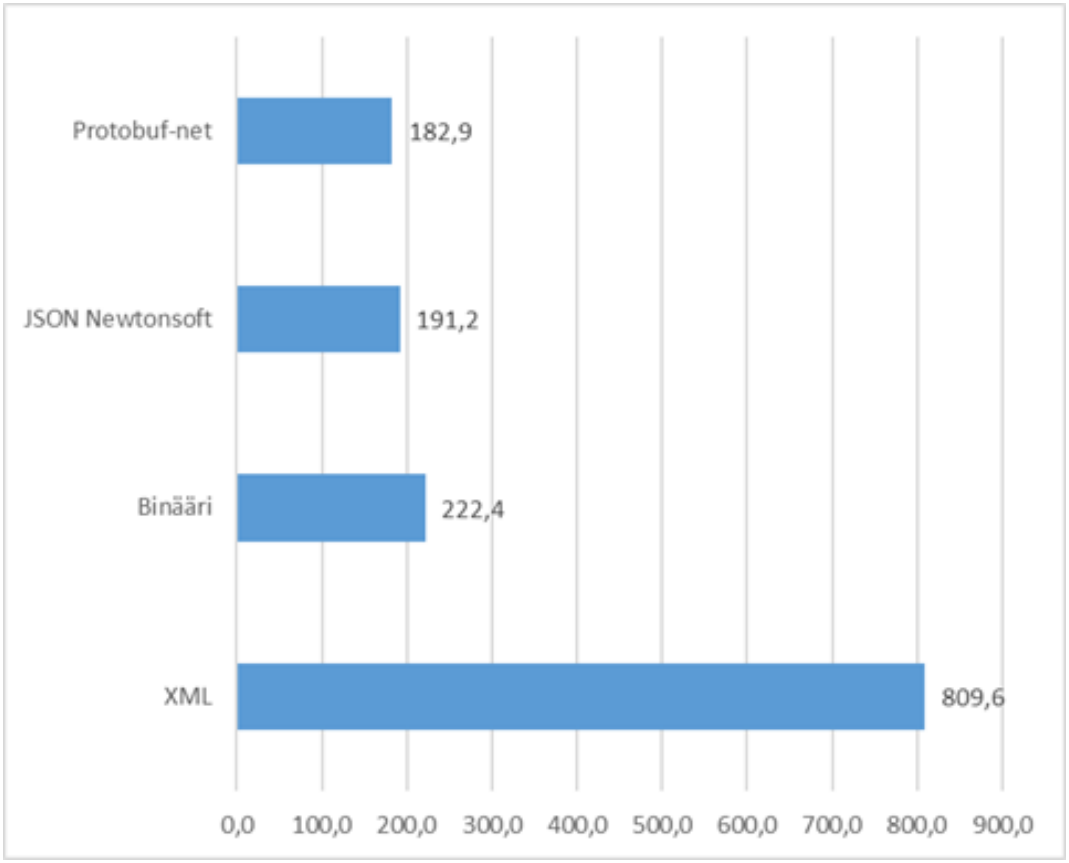


Kuvio 9. Serialisoitujen mallien yhteiskoko levyllä tavuina.

6.4 Lukeminen

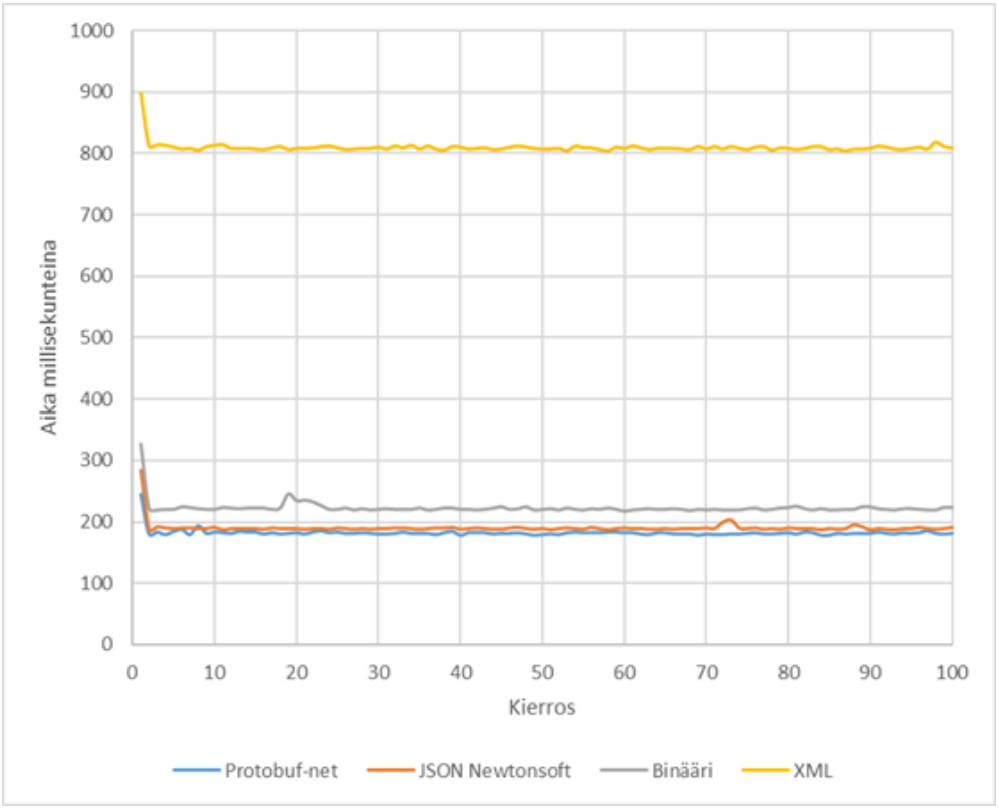
Tässä osiossa on deserialisointitestit purettuna kirjallisesti sekä käyttäen havainnollistavia kuvioita. Kuvioina käytetty on vaaka- ja käyrädiagrammeja. Deserialisointitestissä 500 mallia luettiin kovalevyltä. Testi toistettiin 100 kertaa.

Nopein formaatti testeissä oli Protobuf-net. Toiseksi nopein formaatti oli JSON Newtonsoft, kolmanneksi Unity3D:n binääri ja hitain formaatti oli XML. Kuviossa 10 on esitetty serialisointiformaattien keskimääriset deserialisointiajat vaakadiagrammeina. Protobuf-net-formaatin keskimääräinen deserialisointiaika oli 182,9 millisekuntia. Newtonsoft JSON -formaatin aika oli keskimäärin 191,2 millisekuntia. Unity3D:n binääriformaatin aika oli keskimäärin 222,4 sekuntia ja XML-formaatin aika keskimäärin 809,6 millisekuntia. Suhteutettuna Protobuf-net-formaattiin JSON-formaatti oli 4,6 prosenttia hitaampi. Binääriformaatti oli 21,6 prosenttia hitaampi ja XML-formaatti 342,6 prosenttia hitaampi.

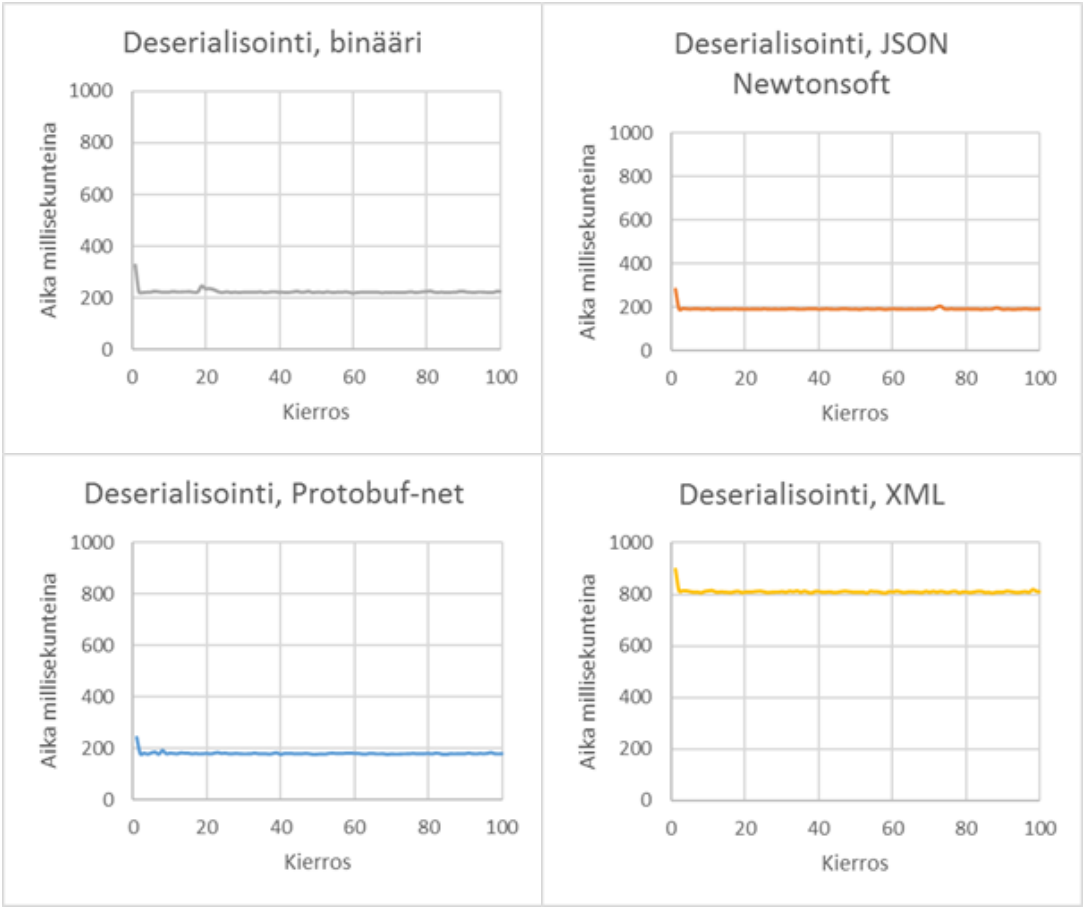


Kuvio 10. Keskimääräinen aika deserialisoinnissa millisekunteina.

Kuviossa 11 on esitettynä deserialisointitestien tulokset käyrädiagrammina, jossa kaikki testatut formaatit ovat yhdessä. Kuviossa 12 formaatit ovat eriteltyinä. Käyrissä näkyvä ensimmäisen deserialisointikerran pidempi kesto johtuu C#-ohjelmointikielen Just-In-Time-kääntäjästä. (Todorov 2013.) Sen sijaan, että ohjelma kääntäisi kaikki tyypit ja metodit käynnistäessä, kääntää se ne vasta kun niitä tarvitaan ensimmäisen kerran.



Kuvio 11. Deserialisointitestien tulokset yhdessä.



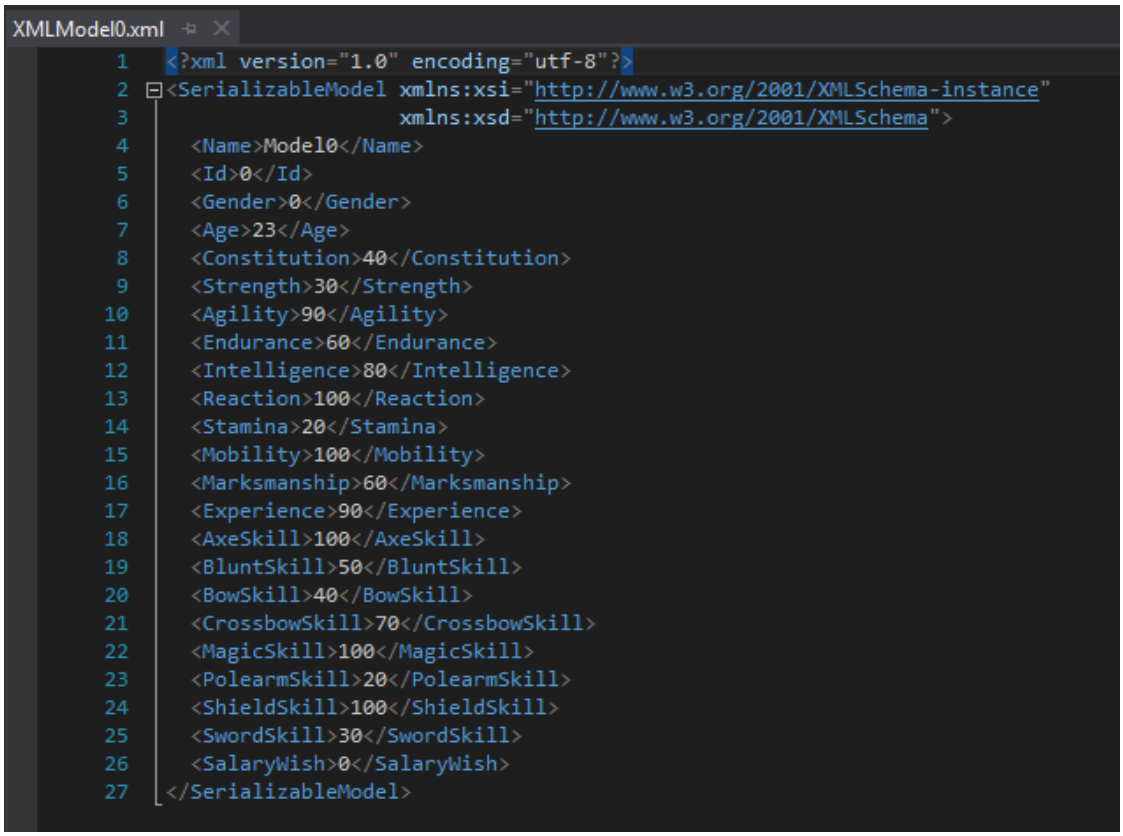
Kuvio 12. Deserialisointitestien tulokset eriteltynä formaattien mukaan.

6.5 Luettavuus

Serialisoidun datan luettavuus on hyvinkin erilainen serialisointitavasta riippuen. Monet formaatit on suunniteltu niin, että data on luettavaa sekä tietokoneelle että ihmissilmälle. Binääriformaatit serialisoivat datan binäärimuotoon ja täten data on useimmiten vain koneluettavaa. Riippuen tekstinlukuohjelmasta ja serialisointiformaatista, voi osa datasta olla luettavissa myös ihmissilmälle. Tässä osiossa arvioidaan suorituskykytesteissä serialisoidun datan luettavuutta.

6.5.1 XML

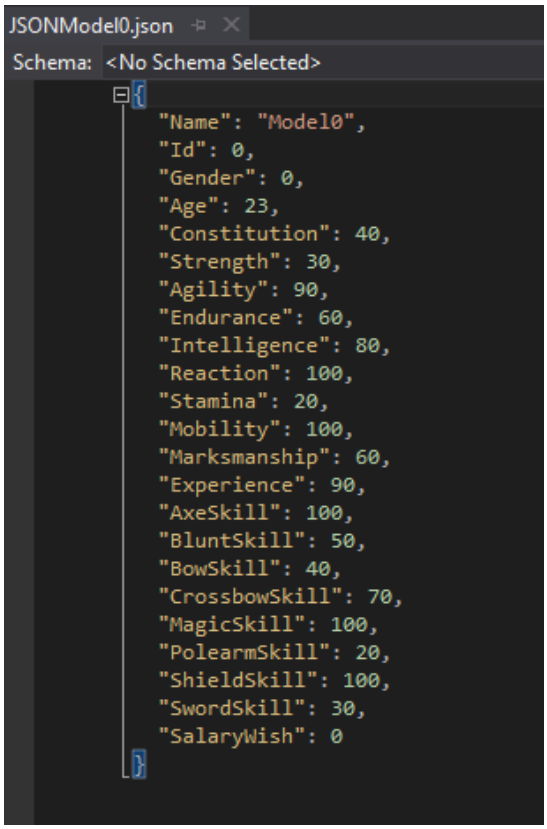
Data on ihmissilmälle luettavissa XML-formaatissa, kuten kuvasta 1 selviää. Serialisoidun mallin muuttujien arvoja on myös helppo muokata suoraan serialisoidusta tiedostosta, jos tarve vaatii.



Kuva 1. Serialisoitu malli XML-formaatissa avattuna Visual Studio -ohjelmointiympäristössä.

6.5.2 JSON

XML-formaatin tapaan, kun malli serialisoidaan JSON-formaatilla, tulee siitä helposti luettavaa ihmissilmälle. Dataa voi myös muokata ilman erillistä kääntäjää yksinkertaisellakin tekstinmuokkausohjelmalla. Kuva 2 on esimerkki serialisoidusta mallista JSON-formaatilla.



Kuva 2. Serialisoitu malli JSON-formaatissa avattuna Visual Studio -ohjelmointiympäristössä.

6.5.3 Binääri

Serialisoidessa Unity3D:n binääriformaattiin ei data ole ihmissilmälle luettavaa suoraan. Jotkin tekstieditorit ja ohjelmistoympäristöt pystyvät kääntämään binäärimuotoista dataa merkkijonoiksi. Kuvassa 3 on binäärimuotoon serialisioitu malli Microsoftin Visual Studio 2015 -ohjelmistoympäristössä. Visual Studiossa data on ilmaistu heksakoodi- ja ASCII-muodossa. Oikeimman puoleisessa sarakkeessa on merkkijonoina serialisoidun mallin kentät. Niiden arvot ovat serialisoinnin johdosta täysin binäärimuodossa.

BinaryModel0.bin

```
00000000  00 01 00 00 00 FF FF FF  FF 01 00 00 00 00 00 00  ................
00000010  00 0C 02 00 00 00 53 53  65 72 69 61 6C 69 7A 61  ......SSerializa
00000020  74 69 6F 6E 4D 65 74 68  6F 64 42 65 6E 63 68 6D  tionMethodBenchm
00000030  61 72 6B 2C 20 56 65 72  73 69 6F 6E 3D 31 2E 30  ark, Version=1.0
00000040  2E 30 2E 30 2C 20 43 75  6C 74 75 72 65 3D 6E 65  .0.0, Culture=ne
00000050  75 74 72 61 6C 2C 20 50  75 62 6C 69 63 4B 65 79  utral, PublicKey
00000060  54 6F 6B 65 6E 3D 6E 75  6C 6C 05 01 00 00 00 2E  Token=null......
00000070  53 65 72 69 61 6C 69 7A  61 74 69 6F 6E 4D 65 74  SerializationMet
00000080  68 6F 64 42 65 6E 63 68  6D 61 72 6B 2E 53 65 72  hodBenchmark.Ser
00000090  69 61 6C 69 7A 61 62 6C  65 4D 6F 64 65 6C 17 00  ializableModel..
000000a0  00 00 04 4E 61 6D 65 02  49 64 06 47 65 6E 64 65  ...Name.Id.Gende
000000b0  72 03 41 67 65 0C 43 6F  6E 73 74 69 74 75 74 69  r.Age.Constituti
000000c0  6F 6E 08 53 74 72 65 6E  67 74 68 07 41 67 69 6C  on.Strength.Agil
000000d0  69 74 79 09 45 6E 64 75  72 61 6E 63 65 0C 49 6E  ity.Endurance.In
000000e0  74 65 6C 6C 69 67 65 6E  63 65 08 52 65 61 63 74  telligence.React
000000f0  69 6F 6E 07 53 74 61 6D  69 6E 61 08 4D 6F 62 69  ion.Stamina.Mobi
00000100  6C 69 74 79 0C 4D 61 72  6B 73 6D 61 6E 73 68 69  lity.Marksmanshi
00000110  70 0A 45 78 70 65 72 69  65 6E 63 65 08 41 78 65  p.Experience.Axe
00000120  53 6B 69 6C 6C 0A 42 6C  75 6E 74 53 6B 69 6C 6C  Skill.BluntSkill
00000130  08 42 6F 77 53 6B 69 6C  6C 0D 43 72 6F 73 73 62  .BowSkill.Crossb
00000140  6F 77 53 6B 69 6C 6C 0A  4D 61 67 69 63 53 6B 69  owSkill.MagicSki
00000150  6C 6C 0C 50 6F 6C 65 61  72 6D 53 6B 69 6C 6C 0B  ll.PolearmSkill.
00000160  53 68 69 65 6C 64 53 6B  69 6C 6C 0A 53 77 6F 72  ShieldSkill.Swor
00000170  64 53 6B 69 6C 6C 0A 53  61 6C 61 72 79 57 69 73  dSkill.SalaryWis
00000180  68 01 00 00 00 00 00 00  00 00 00 00 00 00 00 00  h...............
00000190  00 00 00 00 00 00 00 00  08 08 08 08 08 08 08 08  ................
000001a0  08 08 08 08 08 08 08 08  08 08 08 08 08 08 02 00  ................
000001b0  00 00 06 03 00 00 00 06  4D 6F 64 65 6C 30 00 00  ........Model0..
000001c0  00 00 00 00 00 00 1C 00  00 00 50 00 00 00 46 00  ..........P...F.
000001d0  00 00 32 00 00 00 28 00  00 00 46 00 00 00 32 00  ..2...(...F...2.
000001e0  00 00 14 00 00 00 3C 00  00 00 1E 00 00 00 14 00  ......<.........
000001f0  00 00 1E 00 00 00 3C 00  00 00 5A 00 00 00 50 00  ......<...Z...P.
00000200  00 00 1E 00 00 00 28 00  00 00 46 00 00 00 28 00  ......(...F...(.
00000210  00 00 00 00 00 00 0B                              .......
```

Kuva 3. Serialisoitu malli binääriformaatissa avattuna Visual Studio -ohjelmointiympäristössä.

6.5.4 Protobuf-net

Protobuf-net serialisoituu myös binääriformaattiin, mutta sen luettavuus ihmissilmälle on käytännössä mahdotonta myös edistyneemmillä tekstinmuokkausohjelmilla ja ohjelmistoympäristöillä. Kuvassa 4 näkyy Visual Studiossa avattuna serialisoitu malli Protobuf-net-formaatilla.

ProtobufnetModel0.bin

```
00000000  0A 06 4D 6F 64 65 6C 30  20 1C 28 46 30 14 38 64  ..Model0 .(F0.8d
00000010  40 28 48 1E 50 1E 58 3C  60 3C 68 1E 70 5A 78 46  @(H.P.X<`<h.pZxF
00000020  80 01 46 88 01 1E 90 01  14 98 01 32 A0 01 50 A8  ..F........2..P.
00000030  01 64 B0 01 5A                                    .d..Z
```

Kuva 4. Serialisoitu malli Protobuf-net-formaatissa avattuna Visual Studio -ohjelmointiympäristössä.

6.6 Johtopäätökset

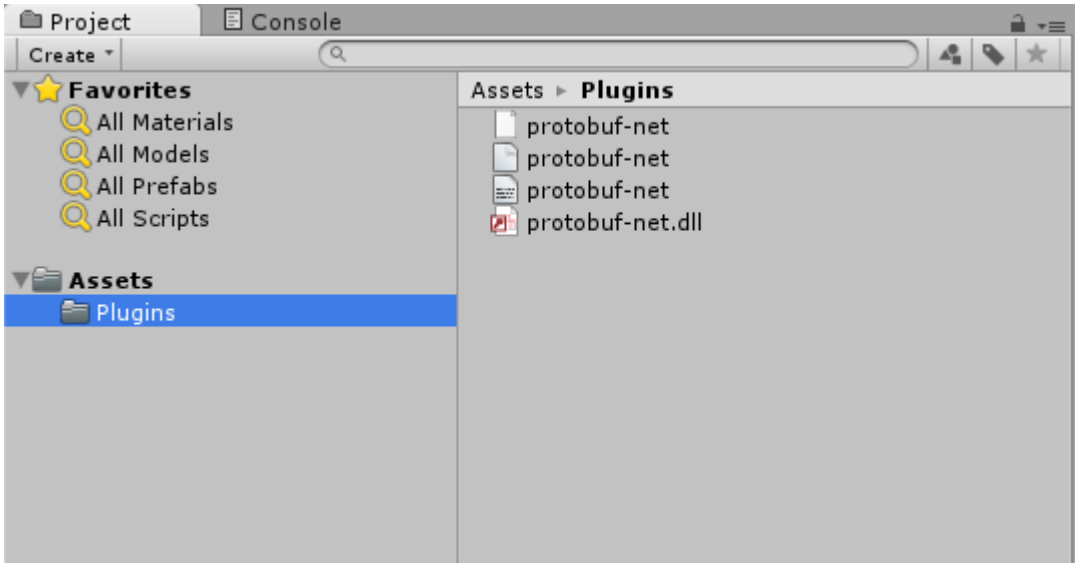
Suorituskykytesteissä testaattiin neljää eri serialisointiformaattia. Formaatteina olivat XML, JSON Newtonsoft, Unity3D:n binääriformaatti ja Protobuf-net. Testattavina osa-alueina olivat nopeus sekä serialisoinnissa että deserialisoinnissa ja tehokkuutta serialisoidun datan pakkaamisessa. Nopeuden ja tehokkuuden lisäksi arivoitiin serialisoitujen tiedostojen luettavuutta.

Serialisointi- ja deserialisointitesteissä nopein formaatti oli Protobuf-net. Se oli myös tehokkuustestin voittaja. Vaikka Protobuf-net oli nopein ja tehokkain, sitä ei ole suunniteltu ihmissilmälle luettavaksi. Serialisointiformaatin valinnasasa on hyvä miettiä mihin tarkoitukseen sitä käytetään. Unityn oma serialisointiformaatti voi olla riittäävä, jos serialisoinnin nopeudella ja tehokkuudella ei ole merkitystä projektissa. Kun serialisoidulta datalta vaaditaan luettavuutta, on syytä käyttää tekstipohjaisia formaatteja kuten XML ja JSON. Protobuf-net-formaatti on testatuista formaateista paras valinta, kun data halutaan saada mahdollisimman pieniksi paketeiksi eikä datan luettavuudella ole merkitystä.

7 TOTEUTUS

7.1 Alkutoimet

Protobuf-net-kirjaston voi ladata Marc Gravellin Google Code -sivustolta. Kirjoitushetkellä viimeisin versio on r668. Ladattava tiedosto sisältää Protobuf-net-tiedostot monelle eri alustalle. Unity-projektiin sopivat tiedostot löytyvät kansion Full alikansiosta unity. Kansio sisältää kolme tiedostoa: protobuf-net.xml, protobuf-net.pbd ja protobuf-net.dll. Tiedostot on vietävä Unity-projektiin. Hyvän käytännön mukaan ulkoiset kirjastot sijoitetaan Plugins-kansioon. Kun Protobuf-net-tiedostot on viety projektiin, voidaan Protobuf-net-kirjastoa käyttää C#-skripteissä.



Kuva 5. Protobuf-net-kirjasto on hyvä asettaa Plugins-kansioon Unity3D-kehitys-ympäristössä.

7.2 Serialisoitavan tiedon määrittäminen

Tässä osiossa on havainnollistettu Protobuf-net-formaatin käyttämistä koodiesimerkein. Protobuf-net-kirjaston käyttäminen onnistuu lisäämällä C#-skriptin alkuun merkintä ”using ProtoBuf”. Jotta Protobuf-net voi tietää, mitä dataa käyttäjä haluaa serialisoitavan, ovat serialisoitavat luokat merkittävä ProtoContract-tagilla samaan tapaan kuin muissakin sopimuspohjaisissa (contract based) serialisointiformaateissa. Luokan kenttiin lisätään ProtoMember-tunnisteet, jotka sisältävät nollaa suuremman kokonaislukuindeksin. Tämän indeksin on oltava myös yksilöllinen luokan sisällä. Jos luokasta löytyy samalla indeksillä olevia serialisoitavia kenttiä, ei Protbuf-net tiedä kuinka se pitäisi serialisoida.

Kaikkia kenttiä ei ole välttämätöntä merkitä ProtoMember-tunnisteilla. Vain ne on merkittävä, jotka halutaan serialisoida.

Koodiesimerkissä 4 on esitettynä WarriorPerkModel-luokka, joka on merkitty ProtoContract-tunnistella ja sen serialisoitavat kentät on merkitty ProtoMember-tunnisteilla ja yksilöllisillä kokonaislukuindekseillä.

```
using ProtoBuf;
…

[ProtoContract]
public class WarriorPerkModel : IWarriorPerkModel
{
    [ProtoMember(1)] public int Id { get; set; }
    [ProtoMember(2)] public string Name { get; set; }
    [ProtoMember(3)] public string Description { get; set; }
    [ProtoMember(4)] public List<PerkCondition> Conditions { get; set; }
    [ProtoMember(5)] public Enumerations.PerkTrigger Trigger { get; set; }
    [ProtoMember(6)] public Enumerations.PerkActivationType ActivationType { get; set; }
    [ProtoMember(7)] public List<ILastingEffectModel> LastingEffects { get; set; }
    [ProtoMember(8)] public List<RestrictionVo> Restrictions { get; set; }
    [ProtoMember(9)] public List<IInstantEffectModel> InstantEffects { get; set; }
    [ProtoMember(10)] public string Icon { get; set; }
}
```

Koodiesimerkki 4. WarriorPerkModel luokka, joka on merkitty ProtoContract-tagilla ja sen kentät somistettu ProtoMember-tageilla.

Jos luokkia halutaan serialisoida periytyvästi tai rajapintaluokkien avulla, on niihin merkittävä ProtoContract-tunnisteen lisäksi ProtoInclude-tunniste. ProtoInclude-tunnisteeseen määritellään luokkaa perivät luokat tai rajapintaluokkien tapauksessa sitä toteuttavat luokat. ProtoInclude-tunniste on lisättävä jokaisella luokalle, jotka käyttävät kyseistä luokkaa. Luokan lisäksi ProtoInclude-tunnisteessa on oltava yksilöllinen tunnistenumero, joka ei myöskään saa olla sama yläluokan ProtoMember-tunnisteiden kanssa. (Stack Overflow: What does the ProtoInclude attribute mean (int protobuf-net) 2009.) Periytyminen on havainnollistettuna koodiesimerkeissä 5, 6 ja 7.

Koodiesimerkissä 5 rajapintaluokkaa IEquipmentModel toteuttaa kolme muuta rajapintaluokkaa: IArmorModel, IShieldModel ja IWeaponModel. Esimerkissä näkyy ProtoInclude-tunnisteet, jotka sisältävät yksilöllisen tunnistenumeron ja IEquipmentModel-rajapintaluokkaa toteuttavat rajapintaluokat. Koodiesimerkissä 6 on näkyvillä IEquipmentModel-rajapintaluokkaa toteuttava rajapintaluokka IWeaponModel. IWeaponModel sisältää ProtoContract -ja ProtoInclude-tunnisteet. ProtoInclude-tunnisteeseen on merkitty WeaponModel-luokka, joka toteuttaa IWeaponModel-rajapintaa. Koodiesimerkissä 7 on WeaponModel-luokka. Luokan

serialisoitaviin kenttiin on merkitty ProtoMember-tunnisteet, jotka sisältävät yksilölliset indeksinumeroinnit.

```
using ProtoBuf;
…

[ProtoContract]
[ProtoInclude(1001, typeof(IArmorModel))]
[ProtoInclude(1002, typeof(IShieldModel))]
[ProtoInclude(1003, typeof(IWeaponModel))]
public interface IEquipmentModel
{
    string Name { get; set; }
    int WarriorId { get; set; }
    int TeamId { get; set; }
    float Condition { get; set; }
    int Price { get; set; }
    int EquipmentType { get; set; }
    int TypeId { get; set; }
    int Id { get; set; }
    int Level { get; set; }
    Enumerations.Roles RoleRestriction { get; set; }
    List<IEffectModel> ModifierEffects { get; set; }
    string IconName { get; set; }
}
```

Koodiesimerkki 5. ProtoInclude-tunnisteet IEquipmentModelissa

```
using ProtoBuf;
…

[ProtoContract]
[ProtoInclude(1100, typeof(WeaponModel))]
public interface IWeaponModel : IEquipmentModel
{
    Enumerations.WeaponType WeaponType { get; set; }
    Enumerations.WeaponCategory WeaponCategory { get; }
    string PrefabName { get; set; }
    int AttackRange { get; set; }
    int UnavailableAttackRange { get; set; }
    int StrengthRequirement { get; set; }
    Enumerations.WeaponSkillType Skill { get; set; }
    Dictionary<int, int> DamageRating { get; set; }
    …
}
```

Koodiesimerkki 6. Rajapintaluokka IWeaponModel. Luokkaa on typistetty havainnollistamistarkoituksessa.

```
using ProtoBuf;
…

[ProtoContract]
public class WeaponModel : IWeaponModel
{
    [ProtoMember(1)] public string Name { get; set; }
    public int WarriorId { get; set; }
    public int TeamId { get; set; }
    [ProtoMember(2)] public int EquipmentType { get; set; }
    [ProtoMember(3)] public float Condition { get; set; }
    [ProtoMember(4)] public int Price { get; set; }
    [ProtoMember(5)] public int TypeId { get; set; }
    public int Id { get; set; }
    public int Level { get { return _level; } set { _level = value; } }
    private int _level = 1;
    [ProtoMember(6)] public Enumerations.Roles RoleRestriction { get; set; }
    [ProtoMember(7)] public List<IEffectModel> ModifierEffects { get; set; }
    public string IconName { get; set; }
    [ProtoMember(8)] public Dictionary<int, int> DamageRating { get; set; }
    [ProtoMember(9)] public Enumerations.WeaponType WeaponType { get; set; }
    public string PrefabName { get; set; }
    [ProtoMember(10)] public int AttackRange { get; set; }
    [ProtoMember(11)] public int UnavailableAttackRange { get; set; }
    [ProtoMember(12)] public int StrengthRequirement { get; set; }
    [ProtoMember(13)] public Enumerations.WeaponSkillType Skill { get; set; }
}
```

Koodiesimerkki 7. ProtoMember-tunnisteet jokaisessa kentässä, jotka halutaan serialisoida luokassa WeaponModel.

7.3 Tiedon serialisointi ja tallentaminen

Challengers of Khalean -pelin kehitystyössä tavoite oli toteuttaa käyttörajapinta Protobuf-netille niin, että se on helppokäyttöinen eikä Protobuf-net palvelun käyttäjän tarvitse käytännössä osata tai tietää itse siitä mitään. Tähän tavoitteeseen pääsemiseksi luotiin Protobuf-net-palvelu StrangeIoC-viitekehyksen mukaisesti, jota on mahdollista käyttää kaikissa projektin MVCS-konteksteissa. Jokaiselle serialisoitavalle luokalle löytyy rajapinnan takaa oma metodi, joka serialisoi ja tallentaa sen määrättyyn kansioon projektissa. Kaikki serialisointimetodit toimivat geneerisen metodin avulla, joka käyttää määritetyn tyypin perusteella oikeaa serialisointimetodia. Koodiesimerkissä 8 on esitettynä rajapinta Challengers of Khalea -pelissä käytettävä Protobuf-net-palvelu. Tiedostopolku on määritetty palvelun sisään ja se on aina suhteessa projektin sijaintiin. Jokainen serialisoitu malli tallennetaan tyypin mukaan omiin kansioihin.

```
using System.Collections.Generic;

namespace Assets.GloryAssets.Scripts.Base.Interfaces
{
    public interface IProtobufIOService
    {
        bool Serialize<T>(string fileName, T model);

        IEnumerable<T> Deserialize<T>();

        T Deserialize<T>(string fileName);
    }
}
```

Koodiesimerkki 8. Protobuf-net-palvelun rajapintaluokka.

7.4 Tiedon deserialisointi ja lukeminen

Kun käyttäjä haluaa lukea levylle tallennettua tietoa, tulee hänen kutsua Protobuf-net palvelun geneeristä Deserialize-metodia Metodi palauttaa geneerisen IEnumerator<T>-kokoelman, joka on määritettyä tyyppiä ”T”. Metodia voi käyttää myös yhden tiedoston deserialisoimiseen antamalla metodille lisäksi merkkijono-tyyppisen parametrin "fileName”. Jos esimerkiksi käyttäjä haluaa ladata levyltä kaikki varustemallit, kutsuu hän metodia koodiesimerkissä 9 havainnollistetulla tavalla. Esimerkissä deserialisoidaan Protobuf-net-palvelun avulla kaikki tyyppiä IEquipmentModel löytyvät mallit ja asetetaan ne muuttujaan equipmentModels.

```
var equipmentModels = ProtobufIOService.Deserialize<IEquipmentModel>();
```

Koodiesimerkki 9. Geneerisen deserialisointimetodin kutsuminen.

7.5 Palvelun laajentaminen

Protobuf-net-palvelulla voidaan serialisoida vain siihen määritettyjä tyyppejä. Projektin laajetessa tulee myös uusia serialisoitavia tyyppejä, jolloin palveluakin on laajennettava. Tässä osiossa on purettu palvelun laajentamisen vaiheet koodiesimerkkien avulla.

Koodiesimerkissä 10 on osa Protobuf-net-palvelun geneerisestä Serialize-metodista. Metodissa on ehtolauseet jokaiselle serialisoitavalle tyypille. Tähän rakenteeseen tulee lisätä uusi ehto, joka sisältää tarkistuksen uudesta tyypistä. Tarkistuksen lisäksi on kirjoitettava

apumetodi tyypin serialisoimiseen. Koodiesimerkissä 11 on esimerkki serialisoinnin apumetodista. Metodin voi suoraan kopioida luodakseen tuen uudelle tyypille. Esimerkkiin on korostettu kohdat, jotka täytyy uudelleen nimetä, jos metodi kopioidaan.

```
using ProtoBuf;
...

public class ProtobufIOService : IProtobufIOService
{
...
   public bool Serialize<T>(string fileName, T model)
   {
      try
      {
         if (model is ITeamModel)
         {
            return SerializeTeamModel(fileName, model as ITeamModel);
         }
         if (model is IAbilityModel)
         {
            return SerializeAbilityModel(fileName, model as IAbilityModel);
         }
         if (model is IEquipmentModel)
         {
            return SerializeEquipmentModel(model as IEquipmentModel, fileName);
         }
         if (model is IWarriorModel)
         {
            return SerializeWarriorModel(fileName, model as IWarriorModel);
         }

         ...

         throw new Exception("Type not recognized.");
      }
      catch (Exception e)
      {
         Debug.Log(e.ToString());
         return false;
      }
   }

...
}
```

Koodiesimerkki 10. Protobuf-net-palvelun Serialize-metodi. Havainnollistamisen vuoksi osa koodista jätetty pois.

```
using ProtoBuf;
...

public class ProtobufIOService : IProtobufIOService
{
...
   private bool SerializeTeamModel(string fileName, ITeamModel teamModel)
   {
      try
      {
         var modelFilePath = SerializationSettings.FilePath
                                        + Sep // System Separator char
                                        + "TeamModels"
                                        + Sep;
         Directory.CreateDirectory(modelFilePath);

         using (var file = File.Create(modelFilePath
                                        + fileName
                                        + ".bin"))
         {
            Serializer.Serialize(file, teamModel);
            return true;
         }
         throw new Exception("File path not set");
      }
      catch (Exception e)
      {
         Debug.Log(e.ToString());
         return false;
      }
   }

...
}
```

Koodiesimerkki 11. Esimerkki serialisoinnin apumetodista. Korostetut kohdat esimerkissä tulee vaihtaa laajennettaessa.

Laajentaminen täytyy tehdä myös geneeriselle Deserialize-metodille. Se on Serialize-metodia vastaavanlainen ja käyttää omia apumetodejaan. Esimerkissä 12 on havainnollistettuna deserialisoinnin apumetodi. Esimerkkiin on korostettu uudelleen nimettävät kohdat.

```
using ProtoBuf;
...

public class ProtobufIOService : IProtobufIOService
{
...

      private IEnumerable<ITeamModel> DeserializeTeamModels()
      {
            var models = new List<ITeamModel>();
            try
            {
                  string[] fileArray = Directory.GetFiles(SerializationSettings.FilePath
                                                                  + Sep
                                                                  + "TeamModels", "*.bin");
                  if (fileArray.Length != 0)
                  {
                        foreach (string s in fileArray)
                        {
                              using (var file = File.OpenRead(s))
                              {
                                    var model = Serializer.Deserialize<ITeamModel>(file);
                                    models.Add(model);
                              }
                        }
            }
                  return models.AsEnumerable();
            }
            catch (Exception e)
            {
                  Debug.Log(e.ToString());
                  throw;
            }
      }

...
}
```

Koodiesimerkki 12. Esimerkki deserialisoinnin apumetodista. Korostetut kohdat esimerkissä tulee vaihtaa laajennettaessa.

8 POHDINTA

Opinnäytetyöni tavoitteena oli toteuttaa helppokäyttöinen ja laajennettava ratkaisu datan serialisoinnille Protobuf-net-formaatilla toimeksiantajayrityksen peliin Challengers of Khalea. Koska projektissa käytettiin StrangeIoC-viitekehystä ja sen tukemaa MVCS-kontekstiarkkitehtuuria, piti serialisointiratkaisusta tehdä oma erillinen palvelunsa. Pääsin mielestäni tavoitteeseen ja Protobuf-net-palvelusta tulikin toimiva kokonaisuus sen käytettävyyden ja laajennettavuuden kannalta. Rajapinta palvelulle on yksinkertainen, eikä se jätä käyttäjälle mahdollisuutta käyttää sitä väärin.

Protobuf-net osoittautui todella tehokkaaksi ja nopeaksi serialisointiformaatiksi suoritus-kykytesteissä. Vaikka Challengers of Khalea onkin yksinpeli, jonka pelaaminen ei vaadi internetyhteyttä, on Protobuf-net mielestäni oiva tapa toteuttaa levylle talletettavan datan serialisointi. Serialisoitujen pakettien koko pysyy hyvin pienenä Protobuf-netin tehokkuuden ansioista, mikä on hyödyllistä, jos dataa pitää siirtää verkon yli tai tiedostokokojen optimointi on suuressa roolissa projektia. Protobuf-net-palvelu on lisäksi helposti laajennettavissa ja projektin edetessä uusille tyypeille on helppo lisätä serialisoinnin tuki. Palvelua voisi myös laajentaa lisäämällä muita hyödyllisiä toimintoja kuten tiedostojen olemassaolo-tarkistuksen tai vaikka automaattisen tiedostojen nimeämiskäytännön.

Jos katsotaan projektia Challengers of Khalea tietoturvan näkökulmasta, tuo Protobuf-net kevyen suojan huijaamista ja pelitiedostojen muuttamista vastaan. Käytettäessä tekstimuotoisia formaatteja kuten XML tai JSON, saattaa pelaaja herkemmin muuntaa tiedostoissa olevia arvoja, jos ne ovat suoraan näkyvillä. Vaikka Protobuf-net binääriformaattina tekee datasta ihmissilmälle lukukelvotonta, on se mahdollista muuttaa luettavaan muotoon. Uskon kuitenkin, että suurin osa pelaajista ei halua nähdä vaivaa serialisoidun datan kääntämiseen.

Yksi harkitsemisen arvoinen kehitysidea Protobuf-net-palvelulle voisi olla tiedostojen kryptaus serialisoinnin lisäksi. Kryptaaminen on käytännössä salakoodaamista, jolla saavutettaisiin tallennettavien tiedostojen parempi salaus. Tällä keinolla pelin tiedostojen muokkaamisen mahdollisuus saataisiin käytännössä poistettua kokonaan käyttäjiltä. Kryptaamisen haittapuolena olisi palvelun toiminnan hidastuminen eikä se ole välttämättä tarpeellista, jos serialisoitava data ei sisällä mitään herkkää materiaalia.

Siitä huolimatta, että Protobuf-net pärjäsi hyvin serialisointitesteissä, ei sen valinta jokaiseen mahdolliseen projektiin ole välttämättä paras ratkaisu. Jos projektissa halutaan saavuttaa serialisoidun datan helppo muokattavuus, ei Protobuf-net sovellu siihen tarkoitukseen. Kuten alaluvussa 6.3 on esitetty, serialisoi Protobuf-net datan binäärimuotoon eikä se siten ole luettavissa ihmissilmälle. Jos Challengers of Khaleaa olisi alun perin kehitetty ilman verkkoyhteyttä pelattavana yksinpelinä, en usko, että serialisointiformaatiksi olisi valittu Protobuf-net-formaattia.

Suunnittelemani suorituskykytestien tulokset ovat viitteellisiä ja testeissä käytettävä aineisto on suunniteltu mukailemaan projektissa Challengers of Khalea käytettävää dataa. Testeistä ei siis käy ilmi, kuinka formaatit suorituvat esimerkiksi todella suurien datamäärien serialisoinnissa. Mielestäni kuitenkin onnistuin testien avulla havainnollistamaan eri serialisointiformaattien eroja ja samalla tuomaan esille niiden hyötyjä ja haittoja.

MVCS-arkkitehtuuri ja StrangeIoC-viitekehys eivät olleet minulle ennestään tuttuja ja datan serialisointiratkaisun luominen niiden mukaan tuottikin ajoittain paljon haasteita. Kun riippuvuuksien vähentäminen ja koodin modularisuus ovat iso osa kehitystyötä, voi yksinkertaiselta kuulostavan ominaisuuden luominen osoittautua työmäärältään moninkertaiseksi. Koodimoduulien välisten riippuvuuksien abstrahointi hämärtää usein ohjelman toiminnan kulun ja virheiden löytämisestä tulee hankalampaa. Ajatusmaailman muuttaminen ei tapahtunutkaan muutamassa päivässä vaan se tapahtui pikkuhiljaa yrityksen ja erehdyksen kautta.

Opinnäytetyöni tekemisen parissa sain ensimmäisen kosketuksen laajemman peliprojektin kehityksestä. Ohjelmointitaitoni Unity3D-pelinkehitysympäristössä ja sen ulkopuolella kehittyivät huomattavasti. Opinnäytetyöni aiheen aniosta sain myös tilaisuuden syventyä datan tallennukseen ja serialisointiin peleissä, mikä on mielestäni mielenkiintoinen aihealue. Koska opinnäytetyöni aiherajaus oli suhteellisen tarkka, toi suorituskykytestit hyvän ja informatiivisen lisän siihen. Aiherajauksen tarkkuudesta huolimatta onnistuin mielestäni selventämään riittävästi projektin kehitystyössä käytettyjä menetelmiä. MVC-arkkitehtuuri ja StrangeIoC-viitekehys ovat suhteellisen laajoja aihe-alueita ja niiden perusteellisempi avaaminen tässä opinnäytetyössä ei mielestäni ollut tarpeellista. Onnistuin mielestäni myös kuvaamaan selkeästi ja yksityiskohtaisesti Protobuf-net-palvelun toteutuksen.

LÄHTEET

Binstock, A. 2013. After XML, JSON: Then What? Tulostettu 23.8.2016 http://www.drdobbs.com/web-development/after-xml-json-then-what/240151851

DB-Engines. Tietokantajärjestelmiä listaava sivusto. Tulostetu 24.8.2016 http://db-engines.com/en/ranking

Dreamloop Games Oy. 2016 Kotisivut. Luettu 12.8.2016. http://www.dreamloop.net/

Fowler, M. 2005. InversionOfControl. Tulostettu 11.8.2016. http://martinfowler.com/bliki/InversionOfControl.html

Google. 2016a. Chrome dokumentaatio: MVC Architecture. Tulostettu 26.8.2016 https://developer.chrome.com/apps/app_frameworks

Google. 2016b. Dokumentaatio kehittäjille: Protocol buffers. Tulostettu 29.8.2016. https://developers.google.com/protocol-buffers/docs/overview

Gravell, M. 2016 Protobuf-net. Tulostettu. https://github.com/mgravell/protobuf-net

Reenskaug, T. 1979. MVC XEROX PARC 1978-1979. Tulostettu 4.7.2016. http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

Ryan, V., Seligman, S. & Lee, R. Sun Microsystems, Inc. 1999. Java RFC 2713. Tulostettu 16.8.2016. https://tools.ietf.org/html/rfc2713

Saikkonen, R. 2012 Ohjelmoinnin peruskurssin laaja oppimäärä. Luento 5: Serialisointi, lisää ohjelmien suunnittelusta, GUI-ohjelmointia. Tulostettu 26.5.2016. https://wiki.aalto.fi/download/attachments/63548818/luento5.pdf?version=1&modificationDate=1329904753000

Shore, J. 2006. Dependency Injection Demystified. Tulostettu 11.8.2016. http://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html

Stack Overflow. 2009. What does the ProtoInclude attribute mean (in protobuf-net). Tulostettu 16.8.2016. http://stackoverflow.com/questions/947666/what-does-the-protoinclude-attribute-mean-in-protobuf-net

Third Motion, Inc. & StrangeIoC. 2015a. StrangeIoC kotisivut. Tulostettu 29.3.2016. http://strangeioc.github.io/strangeioc/

Third Motion, Inc. & StrangeIoC. 2015b. StrangeIoC käyttöopas. Tulostettu 20.6.2016. http://strangeioc.github.io/strangeioc/TheBigStrangeHowTo.html

Third Motion, Inc. & StrangeIoC. 2015c. Kuvio StrangeIoC MVCS-kontekstiarkkitehtuurista. Tulostettu 28.8.2016. https://docs.google.com/document/d/1OV6vOZB6Od9vKmMlJaKkDUXtWsU8dwVSwIxaa5fQwd8/edit

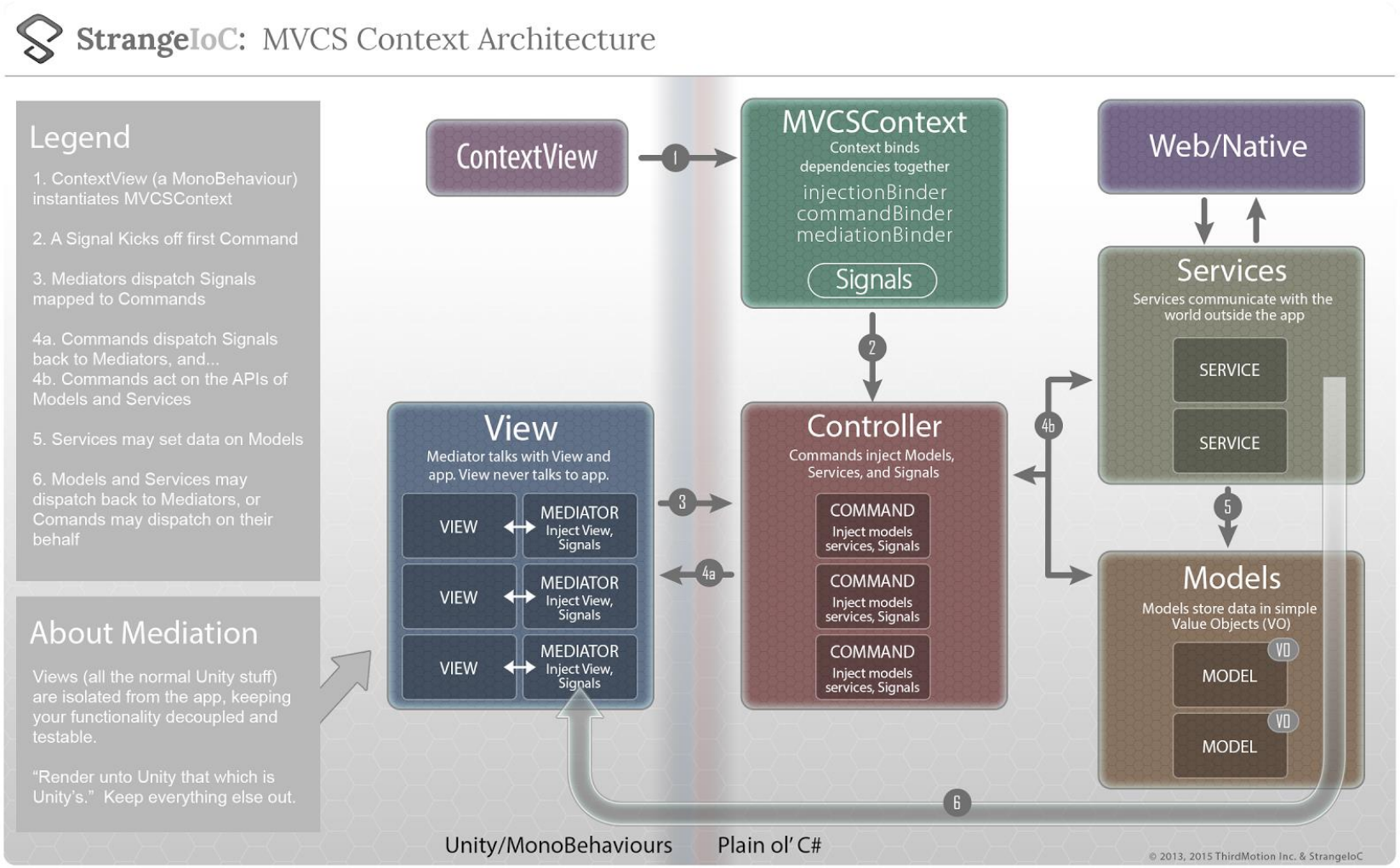
Todorov, T., 28.3.2013. Understanding .NET Just-In-Time Compilation. Tulostettu 29.8.2016. http://www.telerik.com/blogs/understanding-net-just-in-time-compilation

Unity3D. 2016. Scripting API, MonoBehaviour. Tulostettu 24.8.2016.
https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

LIITTEET

Liite 1. StrangeIoC MVCS toimintakaavio.

Lähde: https://docs.google.com/document/d/1OV6vOZB6Od9vKmMlJaKkDUXtWsU8dwVSwIxaa5fQwd8/edit



Liite 2. Esimerkki protokollapuskuri .proto-tiedostosta

```
// See README.txt for information and build instructions.
//
// Note: START and END tags are used in comments to define sections used in
// tutorials.  They are not part of the syntax for Protocol Buffers.
//
// To get an in-depth walkthrough of this file and the related examples, see:
// https://developers.google.com/protocol-buffers/docs/tutorials

// [START declaration]
syntax = "proto3";
package tutorial;
// [END declaration]

// [START java_declaration]
option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";
// [END java_declaration]

// [START csharp_declaration]
option csharp_namespace = "Google.Protobuf.Examples.AddressBook";
// [END csharp_declaration]

// [START messages]
message Person {
  string name = 1;
  int32 id = 2;  // Unique ID number for this person.
  string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
  }

  repeated PhoneNumber phones = 4;
}

// Our address book file is just one of these.
message AddressBook {
  repeated Person people = 1;
}
// [END messages]
```